National Library
of Canada

Canadian Theses Service

Ottawa, Canada
K1A 0N4

Bibliothèque nationale
du Canada

Services des thèses canadiennes

CANADIAN THESES

THÈSES CANADIENNES

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

NL 339 (r. 86/01)

Canada

98

National Library
of Canada

Canadian Theses Division

Ottawa, Canada
K1A 0N4

Bibliothèque nationale
du Canada

Division des thèses canadiennes

0-315-24836-X

PERMISSION TO MICROFILM — AUTORISATION DE MICROFILMER

- Please print or type — Écrire en lettres moulées ou dactylographier

Full Name of Author — Nom complet de l'auteur

AHMED SAMEH MOHAMED

Date of Birth — Date de naissance 09-11-1957	Country of Birth — Lieu de naissance EGYPT

Permanent Address — Résidence fixe

142 Omar Loutfi Street - Sporting Flat 2
Alexandria - EGYPT

Title of Thesis — Titre de la thèse

A proposed Architecture for A 'Mixed-Flow' Database Machine

University — Université

University of Alberta

Degree for which thesis was presented — Grade pour lequel cette thèse fut présentée

Master of Science

Year this degree conferred — Année d'obtention de ce grade 1984	Name of Supervisor — Nom du directeur de thèse Dr. W. W. Armstrong

Permission is hereby granted to the NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

L'autorisation est, par la présente, accordée à la BIBLIOTHÈQUE NATIONALE DU CANADA de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans l'autorisation écrite de l'auteur.

Date October 15th, 1984	Signature Mohamed.

NL-91 (4/77)

The University of Alberta

A PROPOSED ARCHITECTURE FOR A 'MIXED-FLOW' DATABASE MACHINE

by

Ⓒ Ahmed S. Mohamed

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Master of Science

Department of Computing Science

Edmonton, Alberta
Fall, 1984

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Ahmed S. Mohamed

TITLE OF THESIS: A Proposed Architecture for a 'Mixed-Flow' Database Machine

DEGREE FOR WHICH THIS THESIS WAS PRESENTED: Master of Science

YEAR THIS DEGREE GRANTED: Fall 1984

Permission is hereby granted to The University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(Signed) .... Mohamed ....

Permanent Address:

142 Omar Loutfi Street
Alexandria
Egypt

Dated 6 October 1984.

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **A Proposed Architecture for a 'Mixed-Flow' Database Machine** submitted by **Ahmed S. Mohamed** in partial fulfillment of the requirements for the degree of **Master of Science.**

W.W. Armstrong

Supervisor

John Tartar

D. Fisher

T.A. Marsland

Date October 15th, 1984

ABSTRACT

A considerable improvement in processing relational database queries has been shown to be possible using data-driven processing strategies. Moreover, the few implementation trials of data-flow database machines exhibit potential solutions to some of the problems of conventional control-flow database machines.

The intention in this thesis is to follow the data-driven approach in a fashion that avoids the shortcomings of the two extremes: pure control-flow and pure data-flow. A special purpose multi-processor system, the 'mixed-flow' database machine, that lies in between strict control-flow and strict data-flow systems is described. It utilizes a resource pool of specialized functional processors and buffers to embody query trees directly as hardware structures (processor networks), and to execute those queries under a 'mixed-flow' query processing strategy using pipelining and parallel processing.

The 'mixed-flow' query processing strategy introduces the notion of regulating data-flow query processing through planning the flow in advance and adaptively re-allocating resources in the processor network at discrete time-slices. A transformation to a maximum-flow/minimum-cut problem allows planning input flows to different processors in the network and realizes the optimal degree of network pipelining (i.e. optimal given the estimated selectivity factors of the processors). Furthermore, the adaptive re-allocation of resources provides for secure future network operation (close to the optimal planned flow) in addition to resolving existing network bottlenecks.

The 'mixed-flow' strategy is experimentally compared with the pure 'data-flow' strategy and is shown to improve performance with percentages around %25 under normal and heavy loads.

Acknowledgements

I would like to thank my supervisor, W. W. Armstrong, for his constant guidance and support throughout this research. Dr. Armstrong and Hamid Farsi read a rough draft of the thesis and contributed helpful suggestions.

Further thanks are due to my committee members, Dr. J. Tartar, Dr. T. Marsland, Dr. G. Fisher, and Dr. T. Ozsu for the valuable comments on many improvements in this thesis.

Finally, I am deeply grateful to my family, whose contribution over the miles in terms of encouragement, patience and understanding is immeasurable.

TABLE OF CONTENTS

Page

Chapter 1 INTRODUCTION 1

Chapter 2 A SURVEY OF DATABASE MACHINES 10

2.1 Introduction 10

2.2 Database machines research 10

2.3 70's generation database machines 12

2.3.1 RAP 12

2.3.2 CASSM 13

2.3.3 DIRECT 14

2.3.4 DBC 15

2.4 80's generation database machines 16

2.4.1 IDM 16

2.4.2 SPIRIT-III 21

2.4.3 CADAM 22

2.4.4 DIALOG 22

2.5 The consolidation period 23

2.6 Suggested solutions for performance enhancement 28

Chapter 3 Overview of the methodology used in designing 30

3.1 Introduction 30

3.2 Justification 30

3.3 Database dynamics and data-flow query processing strategy 36

3.4 Architecture requirements 43

3.5 specialized functional processors 46
3.5.1 The ALAP associative memory 48
3.5.2 Processing stations 48
3.5.2.1 Join processing station 48
3.5.2.2 Project processor station 56
3.5.2.3 Update processors (Delete and Modify) 60
3.5.2.3.1 Delete processor station 60
3.5.2.3.2 Modify processing station 61
3.5.2.4 Select processor 62
3.6 Data-flow /Mixed-flow architectures 63
3.6.1 Data-flow architectures 65
3.6.2 Mixed-Flow architectures 67
Chapter 4 Mixed-flow query processing strategy 69
4.1 Introduction 69
4.2 Basic query net 69
4.3 General query net (Computation space net) 71
4.4 Selection of processing strategy 76
4.4.1 Mixed-flow query processing strategy 76
4.4.1.1 Reduction to maximum-flow/minimum-cut problem 81
4.4.2 Simulation results 87
Chapter 5 MACHINE ARCHITECTURE 108
5.1 Introduction 108
5.2 Endo-Architecture Description 109
5.3 Machine Instructions and Data Types 119

5.4 Query Execution Example 124

Chapter 6 Conclusion 131

6.1 Contribution and Consequences of the Research 131

6.2 Future work 135

References 137

Appendix A An Airline Reservation system Schema 141

Appendix B A Proof For Net Maximum-flow Theorem 144

Table of Figures

1. Table 1.1: CASSM Characteristics 17

2. Table 1.2: RAP Characteristics 18

3. Table 1.3: DIRECT Characteristics 19

4. Table 1.4:DBC Characteristics 20

5. Figure 3.0: Query Execution in Multi-processor Environments 36

6. Figure 3.1: Stream-Oriented Query net 42

7. Figure 3.2: ALAP associative memory 50

8. Figure 3.3: Join Processing Station Organization 52

9. Figure 3.4: Join Processor functional block diagram 51

10. Figure 3.5: Petri net sequencing diagram for Join processor 55

11. Figure 3.6: Project Processing Station and Operation Scheme 59

12. Figure 3.7: Page Pair Graph for duplicate elimination 61

13. Figure 3.8: Page Pair Graph for Join operation 61

14. Figure 3.10: Petri net sequencing diagram for Select processor 64

15. Figure 4.1: A Computation Space net for an Airline reservation system 78

16. Figure 4.2: Imposing Concurrency Control Mechanisms on a general Query net 78

17. Figure 4.3: Controlled and Uncontrolled Streaming 80

18. Figure 4.4: Transformation from time frame into a capacity frame 87

19. Figure 4.5: Data-Flow work-load assignment 89

20. Figure 4.6: A Mixed-Flow work-load assignment 89

21. Figure 4.7: Optimal net pipelining 90

22. Table 4.1: Transformation Figures 86

23. Figure 4.8: Simulation Sample Queries 93

24. Figure 4.9: Query 1: Data-flow and Mixed-flow response times 94

25. Figure 4.10: Query 2: Data-flow and Mixed-flow response times 95

26. Figure 4.11: Query 3: Data-flow and Mixed-flow response times 96

27. Figure 4.12: Multi-query: Data-flow and Mixed-flow response times 97

28. Figure 4.13: Query 1: Data-flow and Mixed-flow response times 101

29. Figure 4.14: Query 2: Data-flow and Mixed-flow response times 102

30. Figure 4.15: Query 3: Data-flow and Mixed-flow response times 103

31. Figure 4.16: Multi-query: data-flow and Mixed-flow response times 104

32. Figure 4.17: Forward and backward label streaming 107

33. Figure 5.1: The Basic Data-flow Architecture Scheme 111

34. Figure 5.2: Functional Block Diagram of the proposed architecture 115

35. Figure 5.3: Net updating 116

36. Figure 5.4: Instruction execution cycle 122

37. Figure 5.5: Data token types 123

38. Figure 5.6: Operation and Control Packets 125

39. Figure 5.7: Query net example 128

40. Figure 5.8: Processing Stations Query net 129

41. Figure 5.9: Project Processing example 126

CHAPTER 1

INTRODUCTION

As the growth and complexity of technology increases, so does the amount of information processing needed to keep it functioning smoothly. Technology will continue to grow rapidly in the forseeable future, and information processing capabilities must keep up with this technological demand. In an effort to enhance information processing capabilities, dedicated database management systems on conventional general purpose computers are being used more and more both to process and store information. During recent years, however, there has been an increase of doubt about the capabilities of these conventional architectures for efficiently supporting database management tasks.

As a result of the inability of conventional hardware to provide physical data structures that match logical ones, great efforts have been made in database management software support. However, the added weight of this software tends to degrade performance, reliability, etc. [Hsi79]. In order to alleviate this situation and close the "semantic gap" between the logical view of the data and the architecture, special hardware devices ,known as database machines, have been suggested [CHI74] [DeW78] [BaH79] [SNO79] [GoD80] [LHT82].

Since this suggestion, the work on database machines to support modern data management systems has concentrated mostly on the development of new architecdures. Currently, one can name about two dozen such architectures that have been proposed (see for example [HaD81] [BoD80] [Hsi83] for surveys). Early database machines, such as the XDMS [CHI74], involved a general-purpose back-end database computer which used a conventional computer architecture and required a conventional DBMS. The eventual goal in developing the XDMS back-end computer [CHI74] was to off-load database management duties from the host computer. However, the opportunities for performance improvement were very limited for several reasons. The XDMS was designed for the Network model; accordingly, application programs were written in a high-level procedural language. Thus the communication overhead between the host and the XDMS offset any advantages. Another, and more important, limitation of such systems was that the back-end computer would be a conventional Von-Neumann machine which meant that the same shortage of hardware to support database management tasks still existed.

Initially, special purpose architectures employed logic-per-track devices and logic-per-cylinder devices aimed at associatively processing the database where it is stored. However, these associative disks were only effective for simple database operations (e.g, select); they were able to execute these operations in few revolutions of a disk. Since compound operations (e.g, join) required a large number of disk revolutions, these associative disks caused heavy overhead. Consequently, efforts were switched to improving the performance of executing compound operations. Multiprocessor architectures with high-speed memories which are independent of disks have been proposed. That has really improved compound operations performance, but processing data in memories which are independent of disks means staging a large amount of unneeded data from disks to those memories just to start processing. This, again, tends to degrade the performance of 'select' type operations.

By 1980, following the period of architecture 'expansion', came a period of 'consolidation', which involved the analysis of available architectures. The evaluation work carried out by [KIK82] [HaD81] [HaD81] and [ShZ84] reveals a lot of cumbersome architectures (to be discussed in chapter 2); moreover, it asserts the general fact that no one type of current database machine is best for executing all types of queries. Based upon these results two different avenues of research are currently being explored.

The first research avenue was initiated by Goodman [GoD80] in designing the HYPERTREE machine. In this machine, examination of existing database machines led to their classification according to the strategy used to interconnect their various components. The result was two classes: machines that used a simple one-to-one interconnection between processors and memories, such as RAP [SNO79], and machines that used a complex many-to-many strategy, such as DIRECT [BoD81]. As noted, each type of machine executed some operations efficiently. It was thus concluded that a database machine should possess both kinds of communication capabilities. In HYPERTREE, processors are organized as a binary tree, but with each node connected in some regular manner to one of its siblings. Leaf processors are interconnected using the perfect shuffle structure and are connected to disks. The leaf processors are responsible for the execution of the simple 'Select' operations. Thus, they act as data filters to the higher level processors which are responsible for executing the compound operations.

The methodology used in designing HYPERTREE is just an 'add-on' to present architectures or a combination of partially successful designs. These kinds of design philosophies not only increase the complexity of current architectures, but also lead to questionable reliability.

Following the same research avenue, Stonebraker [Sto79] presented several arguments for rejecting other database machine approaches. By contrast, his MUFFIN approach treats a database machine as a special case of distributed databases, and thus reduces the database machine issue to a previously unsolved problem. Similar reduction was carried out in [BeY80] in the design of DIALOG.

Consequently the failure to reach such a 'best' database machine left the case for database machines unproven. Unresolved problems have motivated the belief that the database machine has become an idea whose time has passed [BoD83]. Moreover, some researchers now believe that the database machine notion was not a good idea to start with. Date [Dat83], for example, has completely denied that current machines can pay off for general database applications, and has suggested they be moved to serve in other limited areas for which their capabilities are better suited, such as a level in storage hierarchy or catalog and index support.

Limiting the usefulness of current database machines to only a narrow sphere of applications was also suggested by King [Kin80] who indirectly criticized the idea of database machines when he asserted that the results obtained with system R demonstrated very clearly that exotic hardware (associative memory, etc.) is not required in order to achieve good performance in a relational system.

In contrast to previous opinions, in 1982, a research project known as the fifth-generation computer systems project was started in Japan to further the research and development of the next generation of computers [TrL84] [FeM83]. As suggested by this project, a new generation of computers should evolve from the current research and development of knowledge-base management systems. The key to the implementation of a knowledge-base machine is the underlying database machine.

Consequently, a parallel relational database machine architecture is an appropriate starting point for a knowledge-base machine. Two new types of computing devices are now being developed in Japan: the knowledge-base machine and the problem-solving and inference engine. It was also reported [TrL84] that supercomputers (specially data-flow computers) are considered to be the major focus of the research on these devices.

Only in recent work [LHT82] [BoD80], as more problems have arisen from the development of larger and larger complex architectures, has it been suggested that database machine development is an architectural science in which precise steps or methodologies may be followed to ensure that proper development occurs. It is very likely that these steps will be apparent as a global view of the database requirements is identified.

The second research avenue started from this insight and it has established the fact that the same effort is required for transactions design as for database design [Rol82]. It also points to integration of the dynamic aspects (i.e, the transaction actions on the database, which are conventionally known as operations and events) with data structuring to give a global view of database requirements.

More precisely, it was stated by [BCA83] that both operations and events modeling are useful for:

[1] Indicating dependencies between transactions,i.e, sequences, parallelisms and mutual exclusions.

[2] Defining conceptual schema evaluation rules and describing dynamic integrity constraints.

[3] Providing incremental specifications of data which are needed by each operation.

[4] Providing formal specifications of the interaction with the database for the design of application programs.

With that close look at database dynamics, from such a general perspective, it becomes possible to specify the architectural requirements that are needed for efficiently serving database management tasks. This will facilitate the job of finding the architectural model most closely suited to database dynamics.

The Active Graph database machine [LHT82] is considered to be one pioneering effort in using the previous design methodology. It is one of the few designs that is based on a new data-flow database model. Although the design is not feasible using today's technology, it is obvious that precise steps have been followed during the design process. Here is a brief look at the design.

The new database model is a typical interpretation of a Petri-net model [LeH82] [OzW82], in which each node is viewed as an active entity (data entity constituting the database), with which one processing element is theoretically associated. It is capable of receiving values, arriving asynchronously in the form of tokens, along any of its arcs. Arcs are the representatives of relations interconnecting active nodes. All requests in the proposed model are based on finding paths between member nodes. These requests are injected into the net in the form of restriction tokens (special data elements); which propagate through the net. Nodes satisfying the request are then selected by another type of token called search tokens. Lubmir suggested an interface language and showed how queries could be translated into appropriate sets of restriction tokens. A highly distributed architecture consisting of large independent processing and memory elements was introduced, using VLSI technology to implement the model.

The main point taken from the Active Graph design is that a data-flow database model concept has never before been tried in database processing. In fact, the whole issue of determining what is to be executed in a typical data flow architecture is rather unclear at present [GPK82], and this detracts from the precision of any careful discussion of difficulties in implementation.

This thesis shows that a mixed-flow architecture that lies in between control-flow (on which most database machines had relied) and data-flow architectures would be able to overcome most of the criticisms Garjski faced by data-flow machines [GPK82]. Our purpose in undertaking this task is threefold.

First, it is felt that the second research avenue, design methodology, supports our view of future database machine design. But, it is not possible to satisfy the functional requirements of compound database queries in a pure data-flow architecture environment. The difficulties in operating on large data structures in a data-flow environment are reported in [GPK82].

Secondly, it is possible to follow the data-flow approach intelligently, i.e, direct the data driven nature of the data-flow architecture to the benefit of database management tasks. The objective here is to be sure that things are done in a fashion that avoids the shortcomings of the two extremes: pure control-flow and pure data-flow. The idea of regulating the execution flow is based on the observation that in pure data-flow database machines, queues of partially completed computations are relied upon to keep the machines busy. There are at least two reasons to believe that this would not lead to minimum execution time (particularly in multi-query environments): (1)The need for immediately executing an instruction whose result is critical for the execution of other instructions will always exist. The selection of this instruction among many other ready instructions to be executed is probabilistic, since non-deterministic selection schemes are always employed in current data-flow machines; and (2) The unregulated order of instruction execution will create queues of partially completed computations which absorb some of the parallelism in the queries. We are not aware of any existing data-flow database machine that pays attention to any of these shortcomings.

Thirdly, the design process should start with a study of a computation model for all the high-level operations to be executed by the machine. The results of such a study could then be used to specify the types of services (low-level primitives) required from the hardware. Finally, an architecture which implements these low-level operations should be designed using off-the-shelf technology [BoD81].

This thesis will concern itself with the design of such a mixed-flow architecture. Specifically, a computation model is presented as a modified petri net where each system activity is a sub-net called a query net. Then an implementation for that model based on the principles of data-flow systems is given. The major objective of the implementation is to utilize resource pools of specialized function processors and buffers in order to increase performance, reliability and availability of the architecture. Performance is then enhanced by attacking our mixed-flow architecture from the point of view of studying algorithms and subsystems for achieving 'the guiding of flow' to the benefit of database management. These algorithms are implemented as mechanisms in specific parts of the design. They act as the intelligent heart of the machine.

Finally, it is important to point out that the design is intended primarily for demonstrating the feasibility of using the 'mixed-flow' architecture concept in the database machine area.

Chapter 2 is a review of the research efforts in developing new database machine architectures. Four generic 70's generation architectures that are currently available and in use are surveyed. Then for the 80's generation architectures, four proposals are reviewed with the emphasis on the architecture type and the design methodology. This is followed by investigating the 'consolidation' period which involved performance evaluations of existing designs. Finally, suggested solutions to enhance performance are presented.

Chapter 3 presents an overview of the proposed architecture. This is meant to be an informal description which will provide a basis for understanding the following chapters. This chapter includes the rationale for choosing the 'Computation Space net' as a modeling methodology to formalize representation of both static (data) and the dynamic (operations and events) database requirements. Then the architectural requirements are deduced from the data-driven nature of the model and the rationale for choosing a 'mixed-flow' architecture for the proposed machine is explained.

A Mixed-Flow query processing strategy is proposed in Chapter 4 to be used in the proposed architecture. The strategy makes use of two new algorithms for work-load assignment and resource allocation. Chapter 5 describes the proposed architecture. All the distinctive features in the design are explained in separate subsections.

Chapter 6 discusses future research and presents a final conclusion of the work presented in this thesis.

CHAPTER 2

A SURVEY OF DATABASE MACHINES

2.1. Introduction

In this chapter some past and present trends in database machine development are looked at. Section 2.2 gives a historical perspective on database machine research. Section 2.3 presents four generic 70's architectures, and Section 2.4 examines some of the 80's generation architectures that are currently available. Section 2.5 looks at the evaluation work done on these architectures, and finally, Section 2.6 examines some of the proposed solutions to enhance performance of current database machines.

2.2. Database machines research

As already mentioned in Chapter 1, the efforts in designing database machines were initiated partly in response to the needs expressed by the users, and partly due to the availability of cheap hardware. Although the main objective is to close the 'semantic gap' between the logical view of the data and the architecture, it is not clear whether one wants to eliminate entirely the use of any software support (e.g, indices) in a database machine [BoD81]. DBC is an example of a database machine that uses indexing to reduce the data space to be searched for each query. (This will be explained in section 2.3.4).

Furthermore, the majority of the machines have concentrated on the relational database model, in which query languages are typically nonprocedural, and thus amenable to execution by a number of processors. It is not clear whether parallelism can be used in a similar way to enhance the performance of other older data models. For example, the Network model was designed to optimize the execution of database query programs by requiring the programmer to incorporate access path information into his program. In such programs, access to the database is performed a record at a time using physical links between the stored records, (i.e, inherently sequential order) [BaH79].

Few database machine research efforts are examining the use of new memory technologies for 'intelligent' storage of the database (see for example [ChF80] and [CLW80]); however, none of these efforts have yet culminated in the design of a complete database machine. Most of the research is geared towards the design of chips using VLSI dechnology.

A review of the research efforts in developing complete database architectures reveals that a clear line could be drawn between the 70's generation database machines and the 80's generation machines. The 70's generation 'expansion' period is characterized by its vague architectural concepts which made the machines behave as 'moving targets' when evaluated [HaD81]. Some of the 80's machines are just an 'add-on' to 70's architectures or a combination of partially successful designs; others are based on very deterministic data models and could easily be related to certain architecture types.

This chapter classifies the 70's generation machines into three categories: associative disks (RAP [SNO79] and CASSM [SN79] are examples), 'off-the-disk' parallel machines (DIRECT [DBF80] is an example), and hybrid architectures (DBC [HBB78] is an example).

For 80's machines three classes are reviewed: custom-designed processors (the IDM [Bri] machine), filtering level processors (the SPIRIT-III [Afi81] and CADAM [Sad81] machines) and "distributed databases analogy" processors (the DIALOG [BeY80] machine).

2.3. 70's generation database machines

Four generic architectures from the 70's generation, which are often cited and reveal the variety of approaches during this period, are chosen. In the following classification a brief description of each architecture is included. Table 2-1 presents a comparison of the architectures by examining the following attributes: machine type, place of implementation (on or off the disks), the data model(s) supported and machine primitive capabilities. The table also points out the weaknesses and strengths of each approach.

2.3.1. RAP

The feature that is common to RAP [SNO79] [OSS77]and CASSM is that a query is executed on the disk, usually with the assistance of a single controlling processor. For this reason, such database machines are termed associative disks. The performance of RAP was compared with that of the conventional DBMS [SNO79]. It was shown that for operations that can be processed in linear time on the conventional DBMS (select type), RAP greatly outperforms the conventional system. However, for operations that require non-linear time on the conventional DBMS, RAP performs only marginally better. As mentioned in Chapter 1, a number of 'Off-the-disk' parallel processing database machines have been offered as alternatives to associative disks as a result of this performance review. In RAP, tuples of relations are stored bitwise along tracks. They are augmented with a fixed number of mark bits used to identify result tuples of one operation that are the input to a subsequent operation. Joins are processed as a series of selection subqueries on the larger relation, using the values of the joining attribute in the smaller relation as the selection criterion.

A virtual-memory RAP machine was described as RAP.2 [SNO79]. In this organization,the database resides on a number of conventional mass storage devices. The RAP.2 system consists of a number of cells, each with a pair of tracks. The controller assumes the additional responsibilities of loading the tracks with data to be examined. Each processor can examine only one track at a time. However, while one track is being examined, the second can be loaded under the supervision of the controller. Advantages and disadvantages of RAP will be discussed in Section 2.5; also, in Table 2-1, it is compared with other 70's machines.

2.3.2. CASSM

CASSM [SN79] was designed to support all the three major database models; it used fixed head disks with complex logic in each head. As in RAP, a tuple (or record) is augmented with a fixed number of mark bits that serve the same purpose. Strings are stored only once and pointers to them are used. These pointers are also used for the implementation of databases in other data models (other than relational). A single controller is responsible for distributing instructions to other processors, and collating and processing intermediate results. For example, all processors can perform arithmetic functions locally. The results from each processor are sent to the controller for a collation of the intermediate results. As will be shown in Section 2.5, connecting the processors via a single bus to the controller is one of CASSM's problems.

Joins in CASSM are performed using a hashing scheme. A hash function is applied to the joining attribute of the smaller relation. Then the result is used as an index to a bit array in an auxiliary memory. Associated with the set bit are the attribute values that hashed to that index. Following this, the hash function is applied to the joining attribute of the tuples in the second relation. The result is then checked against the bit array, and if that bit is set, then a match occurs and the attribute, in the second relation, is marked for output and the corresponding bit in a new array is marked, and the joining value saved. In the next step, the hash function, this time, is applied to the first relation, checking the bit position indexed by the hash value in the new array. If the bit was set the values are compared. A match causes the attribute, in the first relation, to be marked for output. In the final step, the marked attributes are collected by the controller which forms the result relation by materializing the join.

As it will be shown in Section 2.5, this auxiliary memory hashing technique is another one of CASSM's drawbacks.

2.3.3. DIRECT

Dewitt [BoD80] felt that RAP (the most advanced and best known database machine during mid 70's) suffered from a number of major shortcomings. One of these was its performance in the execution of complex operations such as join (see Section 2.5). Another was the single instruction multiple data stream nature of its operation (a single instruction from a single program is executed at a time);i.e, it does not support a multi-query environment.

DIRECT was designed to employ general purpose micro-processors that will operate in an multiple instruction multiple data stream mode.

In its original design, it was intended to serve as a back-end database machine to INGRES. The INGRES parser converts all queries into tree format. Leaf nodes in the tree represent operations that are executed on permanent relations in the database. Non-leaf nodes operate on temporary relations produced by their child nodes. Since all operations require, at most, two input relations, and always produce a single output relation, the query tree is binary.

DIRECT consists of a number of processors (called query processors) whose function is to execute operations such as selection, join, etc. The processors are controlled by a controller which distributes instructions and oversees data transfers to the processors. A number of CCD memories serve as distributed caches to the moving head disks. Query processors and CCDs are connected by a cross-point switch that has two important capabilities: any number of processors can read the same CCD device simultaneously, and any two processors can read from any two CCD devices concurrently.

Unlike associative disks, where the processors are physically bound to specific data, the DIRECT controller initiates instructions as soon as resources become available (i.e, allocates the processors to instructions and CCDs to data).

It is important to note here that the idea of allocating resources (memory and processor) to an operation only when the operation is about to be executed, is similar to the resource allocation strategy used in data-flow architectures. In allocating processors to operations, data-flow architectures delay the allocation until the operation has its input operands available and is about to be executed. In control-flow architectures, the processor allocation is made well before the operation is to be executed (This will be elaborated on in Chapter 3).

The cache memory in DIRECT also serves as a temporary storage device for result pages of one operation that are to be used in a subsequent operation. This feature obviates the need for the mark bits used in associative disk machines.

2.3.4. DBC

One of the first designs that incorporated both on-the-disk and off-the-disk processing capabilities was DBC [HBB78]. DBC has two memories: the Mass memory and the Structure memory. The Mass memory uses several moving head disks, with parallel readout capabilities, to store the database. The heads of the disks are connected to a number of processors which perform search operations. The Structure memory is to be constructed out of one of the new technologies (CCD, MBM, etc.) and is used to hold an index. In order to facilitate the use of indices, frequently accessed data is clustered in as few cylinders as possible. The Structure memory is thus used to reduce the data space to be searched by the mass memory.

A controller receives queries and passes them through a number of stages in order to 're-organize' them in a form executable by the structure processor. The structure processor issues search queries to be executed by the mass memory (on-the-disk). The output from the mass memory passes through a security filter and from there to a post-processing unit for performing the complex operations (off-the-disk). The post-processing unit consists of a number of processors interconnected by a uni-directional ring with a single controlling processor that has a communication line to each processor. In executing a complex operation, each processor receives a block of data and communicates some information about the data to the controller. The controller collates the information from all processors, decides on which communication patterns among the processors are necessary to execute the operation, and notifies the processors. Data exchange between processors is through the ring. As will be shown in Section 2.5, it is assumed that the data to be operated on will fit in the memories of all the processors.(There are no paging nor swapping concepts).

2.4. 80's generation database machines

For the 80's generation period, four different proposals are reviewed with the emphasis on the architecture type and the design methodology.

2.4.1. IDM

One of the very few commercially available 80's generation database machines is the IDM(Intelligent Database Machine) [Bri]. It is considered as a type of specialized function architecture similar to CDC6600. It was created with a focus towards the mid-scale databases, and is, therefore, different from the other very large database supporters such as DBC and DIRECT. The level of service that IDM provides is that of basic relational database management operations. Host communication time and software support are minimized at that level. To build IDM with good

Machine	Designer	Type	Implementation	Primitives	Advantages	Problems
CASSM (Context addr segment sequential memory)	U of Florida 72-79	Architecture: Multiple processors direct search (direct search disk tracks). Functional Cellular architecture[L177] Performance Associative processing (bit serial) using marking mechanisms. -Associative disk or logic-per-track.	Disc modification (Read head +processing element logic+ Write head) Designed for hierarchical, but relational can be supported with modifications such as storing all attribute names redundantly and allowing variable length tuples. It does not support the network data model efficiently. It linearize the hierarchy in a preorder sequence that eliminates the need for redundent keys.	A set of data management primitives resid on the CASSM storage along with the data. The host only sends the parameter names ,which are deleted after execution (automatic garbage colection is running all the time and delete the flagged words). The primitives are written in a high level nonprocedural language CASDAL. Three phases for instruction execution (operand fetch, broadcast)+(all cells execute)+(postexecution)+(garbage collection). -Tuples are stored bitwise along each track.	[1]With associative programming it could stand-alone without front end computer [2]Special aggregate functions ,e.g. MIN, MAX, SUM and COUNT are implemented in the special microprocessor. [3]Since only 32 hardware states are involved in one memory revolution , the cell design and the processor program are simple.	[1]Deletion, insertion are complex ;in some cases these operations move entire blocks of words from one cell to another to make room for inserted data or to fill the gap left by a block of deleted data. [2]Restricted size of database limited to disk size. [3]Backup & recovery problems with this continuous read and write cycle. [4]Only character string, pointer and binary integers are allowed as data types. [5]CASSM literature did not describe how to handle the primitive relational operations. [6][Free79] reported that locking on CASSM remains an open problem, since it is not designed for multquery environment.

University of Alberta

Macchine	Designer	Type	Implementation	Primitives	Advantages	Problems
RAP Relational Associative Processor	U of Toronto RAP.1-2 1975-7 RAP.3 1979	Architecture: Multiple processor Indirect search-staging .SIMD (Single Instruction Multiple data stream). .Associative Processing (word parallel) using marking mechanisms. Functional: Logic-Per-Track (Associative Disks). Performance: High speed parallel processing stagger.	Separate from disks. A DMA link is established between the data bus of the RAP controller (for all cells) and the host memory .It supports the relational model.	It has its own RAP commands. The host is required to compile user queries into RAP commands .Each cell performs a number of relational operations directly on its own memory or indirectly on the memory of other cells. .It has automatic garbage collector at each cell. .Performes Join using Cross-mark technique (outer-inner loop). .Some simple updates are done on-the-fly i.e, the database are modified as they pass through a cell's buffer. .Scheduling of queries and maintaining protection security ,and integrity are done by the host. .RAP premitives are: .Mark subset of recs. .Retrive primitives. .Update primitives. .Aggregriate funs. .Insertion and dels. .Control primitives. .Data model defn.	RAP.1 was designed to run as a stand-alone system, whereas RAP.2 and RAP.3 were designed as back ends. .RAP devices can be servived with the conventional disks ,and be used as file caches, in this case there is direct path between conventional disks and RAP so that data would not have to be transfered through the host.	Complex query expression has to be carried out in a series of marking operations with parts of the expression evaluated on each revolution .It may be considered as 'Select-directed' design ,i.e, the selection is the fastest op. it performes. .Fixed size relation tuples and single relation per track (limitation not exist in RAP.3). .Because of the marking scheme, the hardware is naturally locked during an instruction execution. That means no possible multi-query environment could be realized in RAP.

University of Alberta

machine	Designer	Type	Implementation	Primitives	Advantages	Problems
DIRECT	DeWitt U of Wisconsin 1977	Architecture: MIMD (Multiple instruction stream Multiple data stream). .Multiprocessor Indirect search Performance: Distributed multiprocessor architecture. Functional: It may be viewed as a limited distributed database sys. .More functional directed.	Separate from disk. (Back end). A Back end controller is connected to every query processor via DMA interface. .It supports the relational data model.	.A modified version of INGRES in the host splits user's query into one and two variable subqueries, and sends them to DIRECT's back controller which distributes subqueries among query processors and handle the paging service. .Query primitives ops: .Restrict subsets .Project .Join .Modify .Insert .aggregate	.One of the few multi-query environment machines. .It performs both intera-query and inter-query optimization. .The number of processors assigned to the query is dynamically determined by the priority of the query and the size of relations it refers. .Concurrent updates are controlled through address translation tables. .Relation size is not limited by the size of the associative memory, since it works under a paging system. .Compatible w. INGRES. (an existing database system).	.An expensive crossbar switch has been used to connect processors & memories. .Performing poor w.r.t. Select-directed architectures.

University of Alberta

Machine	Designer	Type	Implementation	Primitives	Advantages	Problems
DBC (Database computer).	Ohio State Univ. 1976 Extend. DBC 79.	Architecture: Multiple Processor combined direct+indirect .Processor-per-head architecture [Dewitt81]. Functional: Functionally Specialized Components Performance Associative Processing.	Needs disk modification (cylinders are indvually content addressable). .The Extended DBC uses a parallel transfer disk (parallel-read-write from disks).	.Has its own Command language. .The database command control processor receives user request from the host and converts it into DBC commands. .It directly searchs the data with track processor for simple queries, and stages the data for complex queries [Freeman79] .Special hardware (the key-processor) is used to maintain indices to enhance performance of the Join operation and duplicate elimination. .A controller coordinats the parallel transfer disk loading into memory modules, then assign processor elements along with the special hardware to do any of the primtive operations, which includes: .Search .Project .Join .Addition .Update .Sort .Delete	One of the few machines which off-loads the host from security checking. (security enforcement). .Capable to operate multiple front ends, and very large databases.	.As processor-per-track, additional revolutions may be needed to complete execution of the query if an output buffer overflows .Relying on indices and other data structures to enhance performance.

University of Alberta

cost/performance over a wide range of storage requirements, moving head disks and access method search techniques were used. Having used that slow data retrieval device, the need for powerful processing elements was essential. Special purpose hardware (the database Accelerator) was used. It is a custom-designed pipeline processor that operates at 10 MIPS. The accelerator was designed to execute specific data management subroutines. The major reason for designing this specialized piece of hardware was the observation that most of the execution time of a relational DBMS is typically spent in a very small portion of its code [EpH80]. The IDM also uses information about the behaviour of previously executed queries to cache frequently accessed data. This Database Accelerator board is approximately twice the cost of a microprocessor board but shows a 30 times increase in performance. At present, the IDM system products are designed for IBM PC, DEC VAX users running VMS or UNIX and PDP-11 running UNIX.

2.4.2. SPIRIT-III

SPIRIT-III [Afi81] is a Japanese proposed relational database machine. It combines features of data-staging architectures (such as RAP) with relational algebra execution architecture (such as DIRECT) by attaching refined preprocessing mechanisms of relational algebra operations to data transfer lines connected between hierarchical levels of memory. In addition to this filtering function, it also incorporates a grouping filter into each stage of the memory hierarchy, which partitions relations into subrelations. This grouping operation is implemented using hashing functions to facilitate heavy relational operations. Thus, without the overhead of interprocessor communications, each processor can execute relational algebra operations in parallel on the few subrelations assigned to it. SPIRIT-III performs the join and projection operations in an execution time of O(n/L) (where L is the number of groups). The design is aimed at improving the level of service for both input/output and processing bounded problems. It was claimed, by the designers, that SPIRIT-III represents the third generation relational database machines, enhancing the capabilities of both relational algebra execution and data staging under an integrated concept[Afi81].

2.4.3. CADAM

CADAM (Content Addressable Database Access Machine) [Sad81] is another Japanese database machine under development. Its concern is the efficiency of data transfer between the main memory and the secondary memory. The designers reported a file access mechanism which is adjustable according to database applications (whether the query requires a large number of records, the percentage of the queries having clustering attributes, etc.). Therefore, the physical data access unit (called PAU) was optimized by taking these characteristics into consideration; this led to variable PAU sizes. Like IDM [Bri], an application dependent adjustable cache memory size was introduced in CADAM. The architecture could be considered a refinement of the 70's generation CAFS machine, which serves at the same filtering level for the transferred data between the main memory and the secondary memory. CADAM supports a multi-query environment; moreover, it uses the 'Shortest first' technique for job scheduling. For example, when a number of queries are processed in CADAM, some may require only mapping operations(restriction, projection), but others may require join or set operations, which take much more processing time. In this case, queries requiring only mapping operations of a single relation do not have to wait until a query requiring additional join or set operations is completed.

2.4.4. DIALOG

[BeY80] reported on DIALOG(DIstributed Associative LOGic), a database machine which facilitates both associative and distributed processing. As distributed processing was used in the design, many useful techniques in distributed databases were applied to enhance its performance. A network was proposed which provided a uniform (but costly crossbar switch) medium to connect data modules. Each DIALOG data module consists of a storage device and an associative processor. The data modules are grouped into clusters and these clusters are connected directly with a backend controller. The major functions of that controller include pre-processing and optimizing the queries, looking up the system directory, establishing links between data modules, initiating and scheduling operations within each data module, receiving and buffering output from data modules, and managing the sharing of resources. All the above functions are implemented in software. The module itself consists of four submodules: the physical storage device which contains the database, the selection processing module which processes projection and selection operations, the associative processing module which compares the output from selection processing with the stored search keys(using the Ramamoorthy associative memory), and lastly, the communication processing module which manages the buffer pool and communicates with other data modules in the network. In general the design seems to be promising since most of the results of the older distributed database field could be applied to DIALOG.

2.5. The consolidation period

It is only after a number of designs, that researchers can begin to evaluate their work. An underlying theory, that explains various phenomena, can then emerge. This section investigates some of the recent performance evaluation studies, and presents the following: results of a performance assessment in which six database machine designs were examined by Dewitt and Hawthorn [HaD81] [HaD81], a technological and engineering view of some designs by [Gel80], and finally, an analytical performance model for parallel processing schemes by [KIK82].

[HaD81] showed the difficulties of evaluating the 70's architectures. He asserted that none of the designers had paid the least attention to measuring the performance of his proposed design against other database machines that happened to exist prior to the time of his design proposal. As stated by Dewitt, "..any attempt to compare these designs will often lead to comparing 'apples and oranges'" [HaD81].

As a result of his performance analysis of existing database machines [HaD81], showed that no one type of database machine was best for executing all types of queries; moreover, for several classes of queries, certain database machine designs are actually slower than a database management system on a conventional processor. The same fact was reported by [KIK82], as a result of his analytical performance model for studying the performance of parallel processing database machines.

Another serious deficiency which characterizes a certain class of database machines involves the 'Incomplete hardware designs'. Such hardware design attempts (e.g, CAFS and STARAN) have led to machines that cannot perform all the primitive functions of database management systems. In particular, some of them can support just one of the database management functions in hardware, such as directory processing or data retrieval; others (such as CAFS) cannot adequately support such critical functions as the update functions [HBB78].

Another group of database machine designers proposed machines that are not feasible using today's technology ,and may never become cost effective. [HaD81] expresses his opinion of these designs:

> "These machines can be spotted by claims of join times which are linear (or even less than linear) in the size of the source relations. While we are not saying that research on exotic machines is of no interest, we feel that any machine whose operation requires either as many (or more) processors as tuples in the smaller of the two relations being processed or an associative memory large enough to hold one of the relations, is a machine that will most likely never be feasible."

Other architects have relied on unrealistic assumptions to enhance the performance of their machines. As an example, although DBC uses special hardware to maintain indices to enhance access to the stored data, it is certainly unrealistic to maintain a secondary index for each attribute of each relation. Moreover relying on indices or any data structure to improve the performance of the primitive database operations should be avoided for two reasons: the overhead of maintaining these data structures can be nontrivial in a database machine environment, and ,occasionally a user will perform an operation for which none of the available indices is useful. To avoid that , the designers had to relay on the unrealistic assumption of maintaining a secondary index for each attribute of each relation.

On the other hand, the designers of RAP started with the recognition of the drawbacks of relying on data structure, so they used associative search capability which eliminates the need for access paths. But,unfortunately, they defined RAP relations,and although it has a flat tabular structure, it is not quite relational as defined by Codd. For example, duplicate records are permitted and their existence is not automatically detected. Consequently, introducing this new data model increases the problems involved in the conversion of an existing database management system to a database machine system. Furthermore the use of marking techniques to select the qualifying tuples prevents the possibilities of a multi-query environment in RAP. By using RAP's 'staging' approach, certain relational operations such as natural join, as well as a sequence of operations referring to a large number of relations, would require frequent staging of data. Finally, RAP as a logic-per-track device (associative disk) is a 'select-directed' type of architecture. Given this building block, other relational database operations have been implemented with varying degrees of success.

Two other serious problems with the processor-per-track architectures were reported by [FrB79]. The first one involves the processing on-the-fly concept which limits the amount of processing that can be performed during a single revolution. In addition, some timing problems might arise if multiple records in a hierarchy or network must be examined to determine if a record in the structure is needed. The second problem concerns the backup and recovery. Constantly rewriting the database will increase the number of errors that can occur.

[HaD81] considered the conventional uniprocessor computers (SISD architectures) which run a tuned operating system to satisfy the needs of DBMS, as a class of the available database machines. He showed that these 'machines' are 'software-directed' toward the optimality of all the database management functions. As this class of machines still relies on the capabilities of conventional database management systems, it suffers from the conventional problems of database system software. Datacomputer [Afi81] is an example of this conventional uniprocessor. It provides facilities for data sharing of a centralized database among dissimilar front-end computers in a network environment. Consequently, its performance remains low because of its software-laden nature.

Although the designers of MIMD (Multiple Instructions Multiple Data stream) architecture database machines (such as DIRECT) have their architecture directed towards greater functionality in the processing elements, instead of the 'selection-directed' avenue of the logic-per-track class, they have a poor capability to perform the selection operation as noted in [KIK82]. Besides, the back-end controller might be a bottleneck under certain conditions.

Another point noted by [HaD81] is the level of service that a database machine should support. The main objective is to reduce the number of interactions between the host computer and the database machine for enhancing the performance. The experience gained from XDMS [CHI74] in which a data request is given to the database machine in a procedural language form, proved to be an obstacle in enhancing the performance.

[KIK82] presented an analytical performance model for parallel processing schemes of relational database operations. He showed that for all existing parallel processing schemes employed by current database machines, there is no parallel processing scheme which is best for executing all types of queries. Furthermore, the efficiency of a parallel processing scheme varies with the characteristics of the given query and the contents of the relations to be processed. On the basis of the analytical results of comparing and evaluating three relational database machines parallel processing schemes , Kiyok suggested a new relational database architecture. The architecture is simply a mechanism for selecting an optimal parallel processing scheme through the use of pre-evaluation analytical formulas.

[Gel80] examined the "logic-per-track" class from a technological and engineering point of view, and reported that the viable technology for this class suffers some shortcomings. For example, CCDs require some power-fail strategy, Bubbles had to be ruled out because some simple arithmetic illuminates a bandwidth problem, and although magnetic disks are a viable technology, volatility considerations indicate that care should be taken in using a shift register inside the cell as a part of the data loop. The point here is that we should not rely heavily on technological innovations because of their high rate of change.

In conclusion, it seems that the type of queries places quite specific requirements on database machines. For example, transactions on statistical databases require the database machine to spend more time performing overhead functions such as directory look-up. For databases of that type, a conventional uni-processor DBMS is probably the most cost effective [HaD81]. This may suggest that a database machine like IDM is the best choice in this case, or even, that a DBMS, running on a conventional processor, can search an index or use a hashing function to find the data as quickly, or more quickly, than any existing database machine [BoD81].

On the other hand, for queries that require the DBMS to scan large amounts of data and therefore spend little of its time performing overhead functions, the response time in database machines is significantly better.

Other significant evidence, gained from these evaluations, is the indication that even within each database machine, the total amount of work and the expected response time varies between the best case and worst case. The definition of best case (and worst case) is different for each machine [BoD81]. For example, for INGRES, best case means that the relation is hashed or indexed and the tuples to be retrieved are on as few pages as possible, thus minimizing the amount of input-output to be performed. For DIRECT and RAP machines, best case requires that the data to be scanned reside in the caches at the time the query was initiated. This could happen either because the data was used in a previous query (or a concurrent query for DIRECT); or because a smart prefetching algorithm brought it in from the mass storage devices in anticipation of its use.

It does in fact seem that the case for database machines remains unproven since these research results are not sufficient to serve as an empirical base for a theory.

2.6. Suggested solutions for performance enhancement

[HaD81] suggested various avenues for future research. One is the investigation of a machine which combines the abilities of the 'select-directed' architectures with those of the 'more-functional-directed' architectures. The notion of merging different partially successful architectures has been tried in two different proposals. [FrB79], in his extended DBC design, has merged the DBC original design with the parallel-read out from disk. To some extent, this combines the ability of the 'select-directed' architecture to process selection queries "on-the-fly" with the ability of the multiple processing design to process complex queries. The same notion of combining designs was done by [KIK82]. It is our belief,as mentioned before, that this kind of design philosophy will not only increase the complexity of the architectures, but will also lead to questionable reliability.

In [BeY80], distributed processing was used in the design of DIALOG and many useful techniques for distributed databases were applied to enhance performance. These techniques include query processing, file placement migration rollback and recovery, etc.. As a matter of fact, many of the results of the older field of distributed databases could be used to optimize the processing of queries in DIALOG. The processing of queries using this kind of distributed database analysis is new in the field of database machines. Previously, the intermediate results of subqueries had to be stored in temporary files before they could be re-used (as it is the case in DIRECT). By allowing intermediate results to be piped to their destinations, higher throughput could be achieved. The only problem with DIALOG is its join algorithm; it was incorrectly assumed that the internal buffers of a data module are big enough to hold both source relations. Moreover, it is expected that the performance of the DIALOG join operation will be lower than that of any multiple processor architecture database machine such as DIRECT, since it was not possible, using current DIALOG architecture, to utilize multiple processors in performing the join operation.

[HaD81] suggested that instead of building 'some-operation-directed' architecture for efficiently executing one or two database primitive operations, and developing, afterwards, algorithms to support all the 'other ' database operations using those basic primitives, that a careful examination and analysis of algorithms for all primitive operations be done first.

Dewitt asserts: "Only after these primitives are known and understood, should one attempt to design a machine".

CHAPTER 3

Overview of the methodology used in designing

the proposed architecture

3.1. Introduction

In this chapter, an overview of the proposed design methodology is given. It is intended as a short, informal description which will provide a good perspective on the details of the following chapters. Section 3.2 looks at some justifications for the methodology used in the design. Section 3.3 first investigates the efforts to find complete descriptive models for database dynamics, and then gives a description for a basic data-flow query processing strategy, deduced from database dynamics models and used in the design. Then, Section 3.4 examines two sets of architectural requirements for efficiently serving both database dynamics and tasks and also, the design considerations imposed by those requirements. In Section 3.5, distinct specialized functional parts (processors) for performing database primitives are presented. The decisions for designing these processors are algorithm-dependent. The 'mixed-flow' architecture model is characterized in Section 3.6, and it is argued that its characteristics both match the desired architecture requirements and provide greater flexibility in incorporating both the distinct specialized functional processors and the basic query processing strategy in the body of a database machine.

3.2. Justification

Computer architecture design is a complex and expensive process. In many cases, there is a wide range of types of programs that will run on the machine. Most of the time the types of abstract data structures and the data usage are so enormous as to cause the machine design to be general-purpose. This, however, is not the case for database machines. Both the data structures and the types of operations are well-defined. Moreover, the number of different operations that the machine should support is quite small (all relational algebra operations, for example [DBF80]). If only one primitive relational operation was the objective of a database machine, time would be better spent developing that. As mentioned in Chapter 2, there have been several attempts to build such 'some-operation-directed' database machines, for example, RAP (Selection-directed), DIRECT (More-functional-directed), and CADAM, CAFS (filter-directed). Unfortunately, there are reasons to believe that this methodology would not lead to all types of queries being answered efficiently:

[a] The need for primitive operations, for specific queries, not covered by the database machine will always exist. These primitives perform very poorly when one tries to implement them using the machine-supported operation (e.g, implementing the join as a sequence of selections in RAP) and may become bottlenecks in the machine.

[b] Even the architectures which combine partially successful designs (such as Hypertree and DBC) are very large and complex.

These may also be the reasons behind the difficulty one faces when examining current literature that describes the architectures of the machines in those projects that are still in progress (Hypertree, DBC and RAP). The new descriptions are significantly different from the original design specifications [BoD80]. This is due, in part, to analyses of these machines that revealed the possibilities of improvements by 'adding-on' to the original designs (e.g, incorporation of buffers in the cells of DBC [BaH79]). However, it is also due to the realization, by the designers, that in order to efficiently support some primitive operations, the architecture had to be modified (e.g, RAP.1, RAP.2 and RAP.3).

The fact that there is no one type of database machine best for executing all types of queries seems to us a very predictable consequence of the inappropriate 'operation-directed' design philosophy and also of the inadequate primitive relational operation algorithms. Architecturally speaking, we have to rely on specific machine capability as a basic building block to build our primitive relational operations. Machine capability can be sorting, broadcasting,etc. Once a specific capability is chosen, a machine that provides this capability efficiently can be built. Theoretically speaking, the chosen capability is not guaranteed to provide optimal algorithms for all primitive operations. Consequently, most of the new designs have relied on the methodology of combining capabilities.

Worse than this, the theory is not definite about the superiority, in the domain of a specific primitive operation, of a specific algorithm over others [VaG84] for the whole range of applications. Consequently, the class of machines which tailor microprogrammed algorithms through dynamic programming, automatic programming or evaluation criteria has been established [SNO79] [VaG84].

However, most of the promising database machine proposals and prototype implementations have involved some sort of intelligence in the relative abilities of the processors to act upon processing requirements prior to producing results in order to compensate for this theoretical defect. For example, intelligence is exhibited by:

[a] Collecting a batch of user requests and optimizing the search strategy when an immediate response is not required. (as it is the case in CADAM [Bab79]).

[b] Determining required data in advance for prestaging (as the intelligent page management subsystem in DIRECT [DeW78] and IDM [Bri]) and minimizing the amount of page swapping [MKY81].

[c] Organizing query and data more intelligently by applying correctness-preserving transformations [Bab79].

[d] Using automatic programming techniques so as to minimize query response time and space. That is, moving to and fro between tasks, evaluating the overall effect of refinement decisions utilizing a variety of relational operation algorithms (incorporating microprogramming techniques) [SNO79] [VaG84].

[e] Having feedback loops from later execution phases to earlier phases. This allows transformation steps to be reconsidered during construction when more information details are known, and allows construction steps to be dynamically modified during execution [RaL77].

This supports the belief that the reason that there is no 'best' database machine design is the same reason that a number of database machines require repeated redesigns, or perform some functions poorly: lack of proper design methodology [BoD81].

In this chapter, the method used to design our proposed database machine is described.

It is believed that backing up from the design of any particular database machine and taking a close look at database dynamics will provide us with the architecture requirements necessary for efficiently serving database management. Besides, as the number of primitive operations that the machine must support is quite small (about ten), not only should one consider designing off-the-shelf functional processors for each operation, but should also study the structure of queries to see how the design can be further tailored to meet the user needs.

This thesis presents a new query processing strategy suitable for a multi-processor database machine environment. The strategy exhibits two new algorithms, one for work-load assignment and one for resource re-allocation. To see why we have chosen to deal with these two areas and where these algorithms fit within the overall query processing framework, Figure 3.0 breaks relational query implementation in a multiprocessor environment into three phases: transformation, construction, and execution. Many significant optimization techniques for the first phase could be deduced from the previously mentioned works. The proposed query processing strategy concerns itself with the optimization of the last two phases by presenting the following new features:

[a] It allows net construction to be dynamically modified during execution.

[b] It performs global optimization to system's response time.

[c] It provides a controlled amount of buffer space to exchange data between pipelined operations instead of the storage of large temporary relations.

[d] It provides the base for an optimized architecture through mapping of query structures onto a hardware structure, thus avoiding wasted 'fragments' of resources which are likely to occur in large mainframe architectures.

[e] It allows the opportunity for a self-adjusting architecture, which may provide more consistent response time in a dynamically changing environment.

[f] It establishs the concept of incorporating transaction interrelations into the hardware construction so as to decrease the number of times we have to bring concurrency control mechanisms into play.

The mixed-flow query processing strategy has been taken as a base for designing the underlying mixed-flow database machine.

One of the reasons behind choosing mixed-flow architectures as an architecture type for our database machine is its great capability to serve database dynamics. As mentioned in Chapter 1, the concept of integrating the database dynamic aspects (operations and events) in data structuring is considered to be essential to have a global view of database requirements. Previous database machines did not incorporate the database dynamics in their architectures.

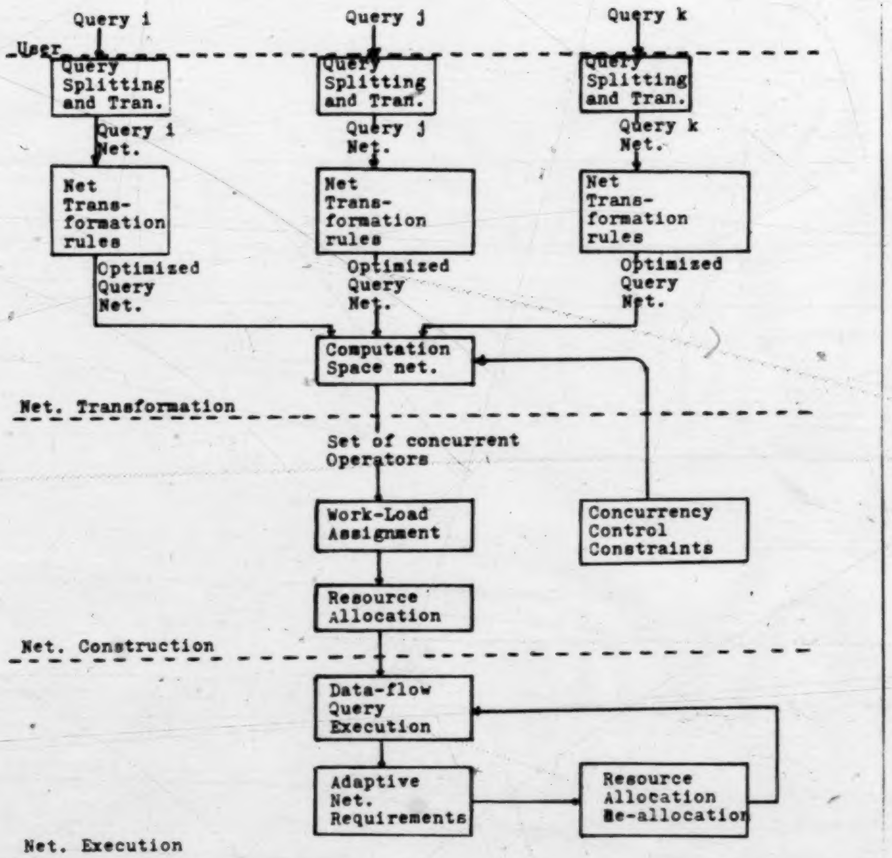

Query i
Query j
Query k
User
Query Splitting and Tran.
Query Splitting and Tran.
Query Splitting and Tran.
Query i Net.
Query j Net.
Query k Net.
Net Trans-formation rules
Net Trans-formation rules
Net Trans-formation rules
Optimized Query Net.
Optimized Query Net.
Optimized Query Net.
Computation Space net.
Net. Transformation
Set of concurrent Operators
Work-Load Assignment
Concurrency Control Constraints
Resource Allocation
Net. Construction
Data-flow Query Execution
Adaptive Net. Requirements
Resource Allocation Re-allocation
Net. Execution

Figure 3.0 Query Execution in Multi-processor Environments

3.3. Database dynamics and data-flow query processing strategy

The efforts to find complete descriptive models for database dynamics can be put into two classes. The first class concerns the specification of transactions through studying semantic relations among data (see for example [BCA83] and [RLR79]). The second class proposes a general description of information systems by taking into consideration the aspects of parallelism and synchronization of transactions [LeH82].

In order to appreciate the notion of database dynamics, here is a close look at an example from each class.

As an example of the first class, the DATAID project [BCA83], presented a new database modeling methodology to formalize representation of both the static (data) and the dynamic (operations and events) database requirements. Three schemata were introduced for data, operations and events. An event schema was defined as a collection of as many event graphs as there are functions (activities)in each environment. An event graph was defined as a description of a specific function (activity) of the environment (e,g, ticket refunds in an airline reservation system). For the representation of event graphs, Petri nets were used, and a complete axiomatic formalism for data-driven representation of transactions was given. Furthermore [AnD83] reported that this formalism must be mapped to a conceptual schema definition language in order to provide data items and structures apt to represent information objects.

In the other class of database dynamics descriptive efforts, [Rol82] presented a general view of a transaction system (a system to describe transactions) in which he tried to integrate the parameters related to the future use of the transaction system and the technical environment in which the system would operate. He presented two levels of transaction modeling. The first level was a conceptual level that allowed an abstract representation of the semantics of the transaction system, both a data schema and a behavioural schema.

The behavioural schema was defined as an atri-alternate graph of some three conceptual aspects :C-objects, C-operations and C-events. The graph highlights the behaviour structure through the interrelations between these concepts. C-objects were the first conceptual aspect,representing a time-consistent aspect of a real-world object class. C-operations were the second aspect, representing a real-world operations class. Finally C-events, the third aspect, represents a real-world events class.

The second level of transaction modeling presented by [Rol82] was a logical level which aimed to take into account the use of both the data system and the transaction system to meet, as well as possible, the user's requirements. This level takes into account the technical constraints of operating.

Following the same ideas, a connection between the modeling of transactions and their processing strategies could be made. Transaction processing deals with the sequencing of transactions, taking into account the aspects of synchronization, parallelism and choice.

Chapter 4 will present rationale for introducing a query model called a 'Query net' which is introduced mainly to facilitate embodying system's queries directly as hardware structures and executing them under a 'mixed-flow' processing strategy using pipelining and parallel processing. The model itself is not new; it is a Petri net form of the old dependency graph, which is being used to study query processing strategies. It offers ,however,a number of facilities which serve our purpose.

Following [LeH82] free-choice Petri nets (see the definitions below) are being used to describe the asynchronous partitioning of queries.

A Query net is defined as an 8-tuple (R,P,A,B,C,g,h,i) where:

R is a free-choice Petri net R(Pr,T,x,y) (see below).

P designates a nonempty set of initial places, P = Pr (Database relations initially reside on these places).

A designates a nonempdy set of elementary transactions of system query.

B designates a set of database integrity constraint predicates.

C designates a set of buffering memories.

g is a function which maps transitions in T' = T into the set of query's elementary transactions A. Tokens of type data-flow through transitions in T'.

h is a function which maps transitions in T-T' into the set B. Tokens of type control-flow through these transitions.

i is a function which maps every place of Pr onto a buffering memory from C.

A Petri net R is defined as a 4-tuple (Pr,T,x,y) where:

Pr is a nonempty set of places.

T is a nonempty set of transitions.

x is a forward incidence function Pr X T → D , where D is the set of natural integers. This function represents the number of tokens that flow from place $Pr_i \in$ Pr to transition $t_i \in$ T (consumed by t_i)

y is a backward incidence function T X Pr → D. D in the set of natural integers and the function represents the number of tokens that flow from t_i to place Pr_i (produced by t_i).

A free-choice Petri net is defined as a Petri net R with the following constraint:

$$\textit{For all } t_i \in T \text{ and } \textit{all}$$

$$P_i \in \ Out(t_i) \rightarrow card(In(P_i)) = 1$$

Where:

Out(t_i) is the set of all places $P_i \in$ P that have an input arc from t_i.

In(P_i) is the set of all transitions $t_i \in$ T, that have an output arc to P_i.

Limiting our attention to only free-choice Petri nets comes naturally with the criterion that any one place has to collect only one type of relational tuples (which constitute an intermediate temporary relation).

Two types of tokens are defined as (1) Data tokens are the entities of the database (either relations, Pages or tuples) (2) Control tokens are Boolean-valued tokens used to direct the flow of the data tokens (by closing and opening the integrity constraint predicate gates).

According to the model, in order to build a 'Query net', first the elementary transactions (subqueries) of the query should be identified. An elementary transaction is a transaction which has the following characteristics: (1)It keeps all the integrity constraints of the database valid after its execution; (2)It works independently of any other transaction in the query; (3) Finally, its different 'pieces' are executed sequentially.

The next step is to impose integrity rules (which represent the query semantics) on these elementary transactions to identify the following:

[a] the transactions which mutually exclude each other;

[b] the critical paths among transactions;

[c] the transactions that are parallel (i.e, can be fired ,i.e executed concurrently), and,

[d] the transactions that are compatible (i.e will never cause any deadlock when executed concurrently).

These relationships define a sort of dependency among the elementary transactions, which enables us to draw a basic query net for the query. This query net explores the maximum parallelism that can be gained in executing the query.

As an example, Figure 3.1, shows the query net for the following query:

"What are the names of the 30$ items which have quality 'A' and are supplied to suppliers who purchased at least one item?".

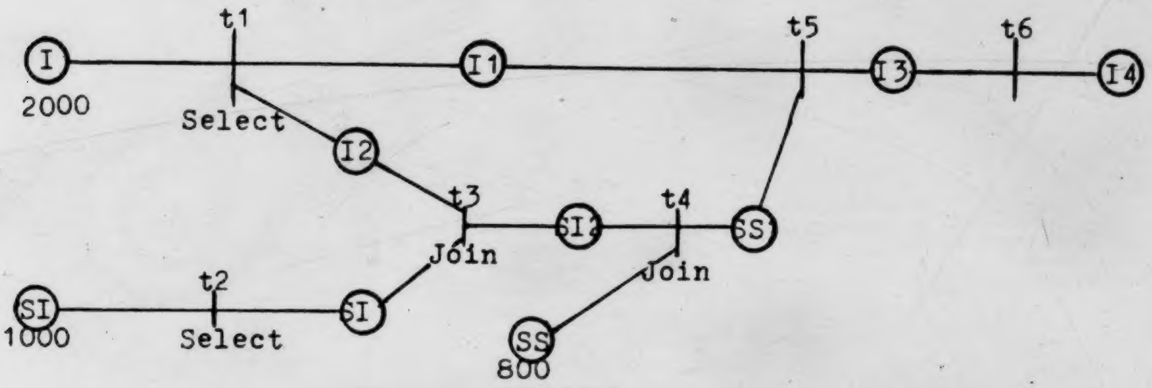
The elementary transactions for this query are:

[1] t1: Select I:price=30$ to get I1 and I2.
[2] t2: Select SI:quality='A' to get SI1.
{3] t3: Join SI1 & I2 (SI1.I no = I2.I no) to get SI2.
[4] t4: Join SS & SI2(SS.S no = SI2.S no) to get SS1.
[5] t5: Join I1 & SS1 (I1.I no = SS1.I no) to get I3.
[6] t6: Project I3(Iname) to get I4.

The Query net in figure 3.1 assumes that relation S contains 500 pages, relation SI contains 1000 pages, relation SS contains 800 pages and relation I contains 2000 pages.

A number of extensions to the basic Query net will be proposed in Chapter 4 so as to allow us to define what is called a "Computation space net", which describes all the functions (activities) of a multi-query database environment as a net of places and transitions. The "Computation space net" is a collection of as many query nets as there are functions (activities) in the environment. Mainly, it will serve as an intermediate language to describe explicitly the maximum parallelism that exists in the target environment.

As the Query net model maps each transition to either an elementary transaction or an integrity constraint predicate and each initial place to a buffering memory that holds a database relation (as in figure 3.1), it is reasonable to consider a page of a relation as the basic unit to be used for scheduling decisions [BoD80]. This means that an elementary transaction can be initiated as soon as at least one page of each participating relation is available.



S:Supplier(S.no,Sname,Saddress)
SI:Sup-Item(S.no,I.no,quality)
I:Item(I.no,I.name,Price)
SS:Supplied(S.no,I.no,quantity)

Figure 3.1: Stream-Oriented Query net

Thus, the model suggests the idea of assigning processors to execute an elementary transaction based on the availability of pages rather than relations on its input arcs. This offers the possibility of a very flexible processor allocation strategy which has a high utilization factor. Furthermore, it becomes possible to distribute processors across all transitions of the net and to pipeline pages of intermediate relations between them. These intermediate relation pages are stored consequently at different places (buffering memories) of the net.

At this point, every part of the Query net is active; this strategy in processing queries is called processing in a data-flow manner [BoD81]. Several benefits can be gained from this strategy; for example, it will reduce page traffic between the buffering memories (net places) and the mass storage devices, because after a page of an intermediate relation is produced by an elementary transaction's processor(s), it will be read by a processor executing the subsequent elementary transition. Another benefit is that it will improve the mean response time in a multi-query environment, since the results of the queries are pushed as much as possible to get early answers.

In fact, there are other advantages, but also disadvantages for this data-flow strategy in processing queries. Discussing them will be postponded till a formal description of the Computation space net model and its extension is given in Chapter 4. For now, here is a discussion of a number of points concerning the implementation of the strategy.

It is clearly understood that there are number of costs associated with the execution of queries in the previous data-flow manner. One of them is the time it takes to perform elementary transactions. Another is the number of processors required at each transition, and the size of buffering at different net places. And, particularly in data-flow computation, it's clear that the communications overhead will substantially affect the cost of executing a query net.

These factors could be alleviated a bit by answering the question of how to assign elementary transactions in a query net to processing units in the machine in a manner which allows three things to occur. First, if at all possible, no concurrency should be lost. Secondly, the solution should use a minimum number of processors. Thirdly, the communication overhead should be reduced as much as possible.

For consistency, there are number of other points that should be investigated first before the previous optimization aspects. For example:

[1] What type of processing units are to be used in executing the elementary transactions?

[2] How the mapping of a query structure onto a hardware structure could be implemented?

[3] What sort of interconnection and scheduling capabilities should the architecture posses to implement that mapping?

To answer these questions and several others, one should establish the architectural requirements which are necessary for efficiently serving the data-flow execution of queries.

3.4. Architecture requirements

This Section will investigate two sets of architecture requirements. The first set represents the requirements necessary for efficiently serving the data-flow execution of queries, which include:

[1] A self-adjusting architecture, which can provide more consistent response time in a dynamically changing environment.

[2] An optimized architecture; through mapping of the current demand on the system onto hardware structures and thus avoids wasted "fragments" of resources which are likely to occur in large mainframe architectures.

[3] An architecture that provides a global view of multi-query environments, where resources are partitioned to form substructures and added/deleted within the inventory of computing power, in response to environment demand.

[4] An architecture able to impose concurrency control mechanisms; since knowing the transactions' interrelations from the Computation space net can decrease the number of times concurrency control mechanisms have to be brought into play. These mechanisms are incorporated into the environment Computation space net (this will be explained in chapter 4).

[5] An architecture able to incorporate dynamic concurrency; since using the pipelining nature of the Computation space net would allow for a new dimension of parallelism in the following sense. In existing distributed systems , there is

some parallelism among transactions being executed at different places because of the distribution of the data. This is called 'Static concurrency'; it refers to the concurrency made possible by data independence. This concurrency is evident at compile time. The new dimension of concurrency that would be available by using our Computation space net is 'Dynamic concurrency' which refers to the concurrent execution of different loop iterations or functional invocations because of the data driven behaviour of the model. Dynamic concurrency is determined at execution time.

So far this section has concentrated on deducing the architectural requirements which are necessary for efficiently serving the data-flow execution of queries. Additional requirements that affect the ultimate design include:

[1] Implementation considerations:

[a] The architecture is required to implement the primitive operations (e.g, Join and Select) and facilitate higher level query optimization.

[b] The architecture should depend on existing technologies in the design. Besides, the design should be able to evolve new technologies that would be available in the near future.

[2] Expansibility issues:

[a] The architecture should be able to support very large databases and should be easily expandable according to the changing user needs.

[3] Performance:

[a] The architecture must have high performance and the cost should be low by replicating a few simple components.

[b] The architecture should have no particular component likely to become a bottleneck under either normal or abnormal operating conditions.

In trying to satisfy the previous sets of architecture requirements on the bases of the data-flow behaviour of the query processing model, the following design considerations have been imposed on candidate architecture models:

[1] The fact that "..different primitives algorithms need different architecture requirements for an efficient execution" reported in [DeW78], suggests distinct specialized functional parts for the performance of specific primitive tasks;

[2] the independent concurrent nature of transactions in the Computation space net suggests a multiple processor environment;

[3] the asynchronous aspect (the production/ consumption nature) of the execution of transactions suggests data-driven behaviour in which the specialized functional processors operate concurrently and communicate with each other asynchronously;

[4] since the intermediate results are local and memory need not be shared, distributed control and memory are possible;

[5] to maintain performance objectives in a variable-load multi-query environment, resource allocation strategies are suggested.

Before proceeding any further with forming the shape of the proposed architecture, this section has to answer first, the question presented in the previous section, which deals with the type of processing units to be used in executing elementary transactions.

The organization of the remainder of this chapter reflects the design methodology described in Section 3.2. A description of the specialized functional processors to be used as off-the-shelf components to execute elementary transactions is given first. Next Section 3.6 characterizes 'mixed-flow' architectures and shows that their characteristics match the desired architecture requirements, and that they can utilize the specialized functional processors in an integrated manner.

3.5. specialized functional processors

The specialized functional processors described in this section are for the relational algebra operations. The operations covered are: select, join, project, delete and modify. It is important to note that once agreed that every one of these operations needs different architecture requirements to be efficiently executed (as mentioned in the design considerations), then decisions to build specialized processors to handle those operations are totally independent from architecture decisions for the overall database machine. That means decisions for efficiently building these primitive processors should be based on performance trade-offs among different algorithms to do the job.

The alternative algorithms for each operator are compared in [BoD80]. The results of this comparison are inconclusive in the sense that no type of algorithm proved to be better under all conditions [VaG84]. Therefore, it is reasonable to choose one class of algorithm (to establish an architecture capability) and proceed from that point. In this thesis the "broadcasting to associative memories" architecture capability has been used. Our primary reason for this choice is simplicity of the control function. Furthermore, for select-type operations, it is clear how the use of associative memories is superior to any other selection algorithm since it can perform a page selection in a constant time. However, this may contradict [HaD81] result which asserts that there is no better performance for select-type operations than processing them on-the-disk. Our point here is that starting to search for the best on-the-disk type of processing is a different line of research. Even if there is a need for such on-the-disk strategy, nothing would be affected in our 'off-the-disc' database machine, other than eliminating the selection operation.

For compound operators (such as join) the best known algorithm is the semi-join algorithm [BoD80].

It is difficult to incorporate any of the semi-join algorithms in the proposed architecture for the following reasons: Whether the machine uses a bit-array or an associative memory to assure the matching of tuples, the next step would be to materialize the join (i.e, concatenate matched tuples from the source relation with the underlying tuple from the target relation). For this step there are currently two available solutions; unfortunately, both rely on hashing techniques to speed up the concatenation. The main shortcoming of using the hashing technique is, since the machine has no way to know the range of the join attribute values, either it has to use more than one hashing function or to have a large enough hashing table to lower the possibilities of collisions. Each of the two solutions requires a large memory, and that requirement contradicts one of the architectural considerations mentioned in Section 3.4, that of having a distributed memory. Moreover, all known Semi-join algorithms claimed possible extensions of their strategies to perform inequality joins and m-way joins[BaH79]. However, it is not clear with a reliance on hashing techniques how to perform inequality joins, or even a join having more than one predicate condition without having a very complex and time consuming algorithm [VaG84].

Two other reasons justify our employing of associative memories in the specialized functional processors:

[1] Many previous database machines have relied on it, such as DBC, STARAN and RELACS [OlB79]. Moreover, STARAN may be considered as a general-purpose associative bit-serial processor, since it uses a 256-bit by 256-word associative memory modules. RELACS, also, is organized as a number of two-dimensional modules whose total storage capacity equals that of a disk track.

[2] The use of associative memories is not extensive in our proposal, since all needed is one page-size associative memory per specialized functional processor.

For all of the above reasons, associative memories have been employed in our specialized functional processors, even though they are normally cost-inefficient.

The specialized processors presented below share a number of general points. First, an associative memory called ALAP, introduced in [FiL77], is being used with every processor. Secondly, relations are organized as a collection of fixed-size pages. The page size should be large enough so that it constitutes an efficient unit of transfer among machine components, but at the same time it should be small enough so that a large number of processors, each examining a few pages, can participate in the execution of the operation.

3.5.1. The ALAP associative memory

The ALAP associative memory is classified as a bit-serial associative memory (the same as the one used in STARAN). Among its 29 arithmetic and logical functions, which can be carried out entirely within the ALAP array, our interest is in its typical associative memory operations: exact match, greater-than-or-equal match and less-than match. Figure 3.2 shows its feature of outputting all matching words one after the other.

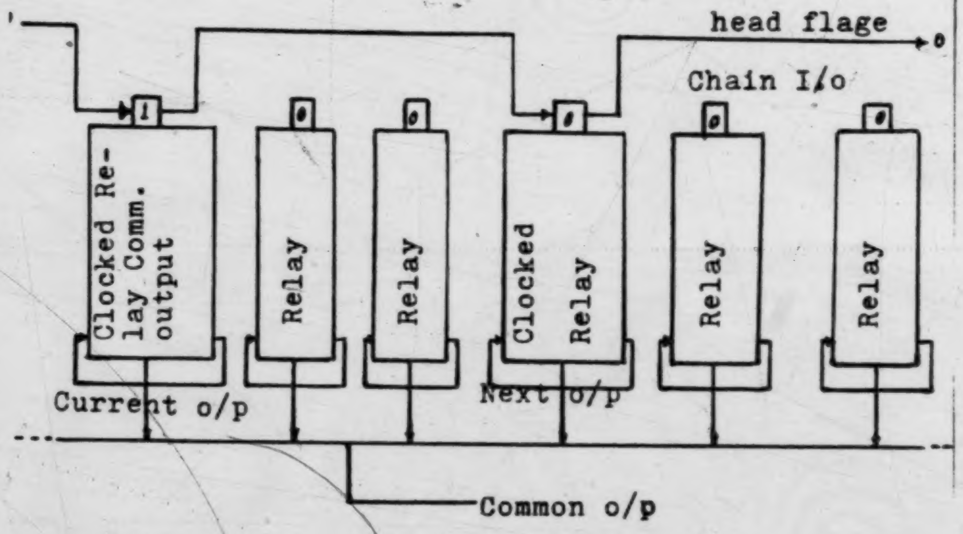
3.5.2. Processing stations

For each of the following functional processors we describe the functional block diagram and theory of operation.

3.5.2.1. Join processing station

Given two relations, R and T, the 'smaller' relation is chosen as the inner relation, and the larger (say R) becomes the outer relation. The join processing station

Figure 3.2 : ALAP associative memory


head flage
Chain I/o
1
0
0
0
0
0
Clocked Relay Comm. output
Relay
Relay
Clocked Relay
Relay
Relay
Current o/p
Next o/p
Common o/p

consists of a 'Ring' of Join processors (as shown in figure 3.3). (The structure of the Join processor will be described soon). The first step in processing the join is for the Join processors to each read a different page of the outer relation. Next, all pages of the inner relation,T, are sequentially broadcast to the processors in the 'Ring'.(That is why the method is called broadcasting to associative memories). As each page of T is received by a processor, it joins the page with its page from R. Joining two pages means: first the join is performed by concatenation, then the result page is written out. The reasons behind using this O(nXm) join algorithm are explained in the next Section (n,m are the sizes of R and T respectively).

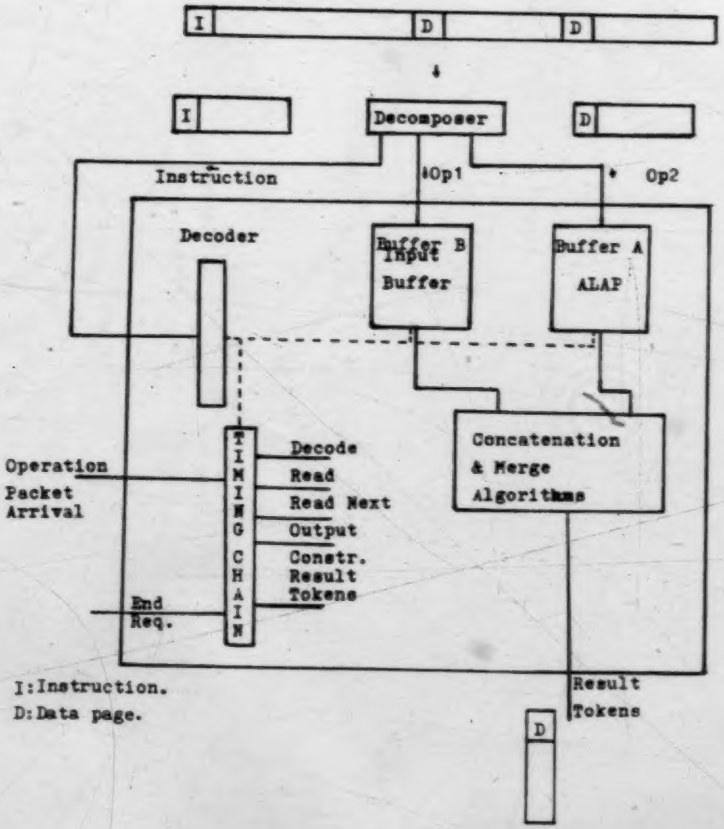
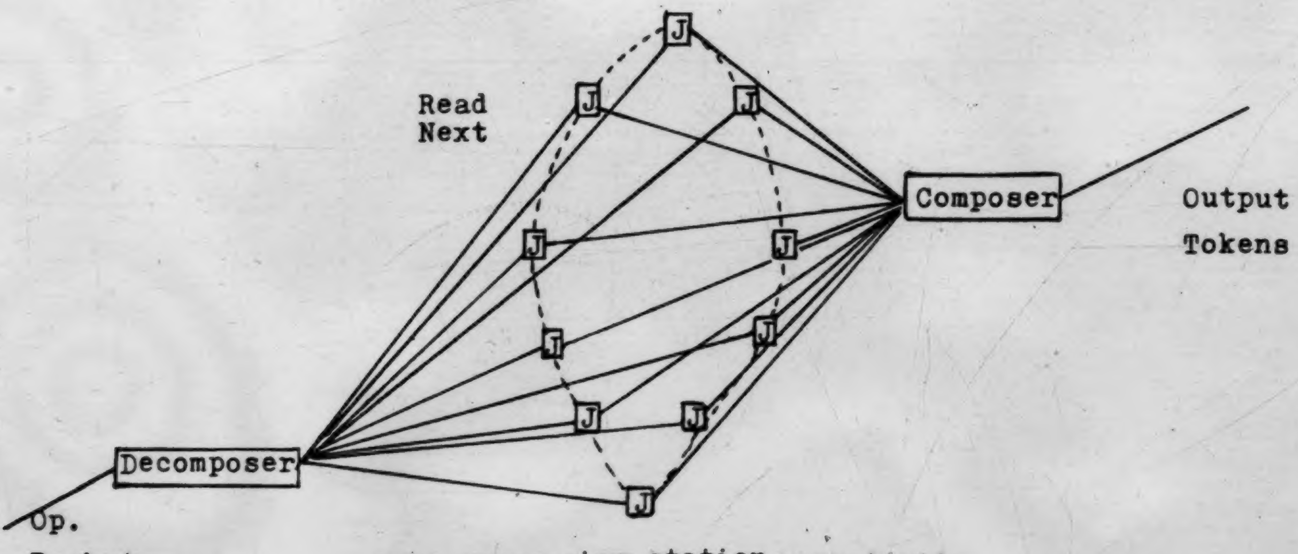

I
D
D
I
Decomposer
D
Instruction
Op1
Op2
Decoder
Buffer B
Input
Buffer
Buffer A
ALAP
Concatenation
& Merge
Algorithms
TIMING CHAIN
Decode
Read
Read Next
Output
Constr.
Result
Tokens
Operation
Packet
Arrival
End
Req.
I:Instruction.
D:Data page.
Result
Tokens
D

Figure 3.4:Join functional block diagram

[a] Functional block diagram:

Figure 3.4 shows a functional block diagram of a join processor. It consists of two input buffers to hold a page from each of the outer and inner relations. Buffer A is an ALAP memory and holds the outer relation page; buffer B is a RAM memory and holds the inner relation page. It also has a concatenation algorithm to materialize the join (i.e, to concatenate the matched tuples (produced by buffer A) from the outer relation page with the underlying tuple of the inner relation page).

As will be shown, the structures of all compound primitive's processors are identical, so only two types of functional processors, one with double buffering and another with single buffering are needed. As any double buffering processor is able to



Join processing station organization

Figure 3.3

perform any of the compound operations (e.g, Join, Project, etc.), so the decoder (in the figure) is used to identify the target operation. In the case of Join, it also has to find the relative positions of the θ operands ($R \infty_{\theta} T$) in which θ is any of the operations (= <> > < >= <=). The Timing-Chain circuit is the controller of the processor, it is synchronized with the timing of the ALAP and used to send the appropriate timing control sequence according to the Petri-net sequencing diagram in figure 3.5.

[b] Operation description:

In the Petri net of Figure 3.5, every transition is associated with a control command from the Timing Chain circuit and each numbered place represents a certain condition to be valid in the processor. The net should be interpreted as follows:

Command :DECODE
Precondition:1,2,3
postcondition:4,5,6

This means conditions 1, 2 and 3 should be satisfied in order for the command DECODE to be issued by the 'Timing Chain' circuit. And once it is issued , some actions should be done by the processor to establish the post conditions 4, 5 and 6.

A signal 'operation packet arrival' to the underlying processor enables its 'Timing Chain' circuit to start issuing its commands according to the Petri net in Figure 3.5. The following is a description of the commands and conditions in the figure:

[1] DECODE :Identifies the operation that needs to be executed by this processor by checking the operation code in the operation packet. In the case of Join, it also has to find the relative positions of θ operands ($T \infty_{\theta} R$).

[2] READ :A control signal issued when both input pages are ready to be read by the processor into its buffers A, B.

[3] READ-NEXT:A control signal issued when the current page in buffer B has been processed and the next page of the broadcasted relation (inner relation) is ready for processing (or ready to be read into buffer B).

[4] END-REQUEST:A message from the underlying processor asking if the results tuples, from the current processed pages, can be routed to its destination(s). This message has to be issued before the completion of the operation so that there will be no delay in routing the result.

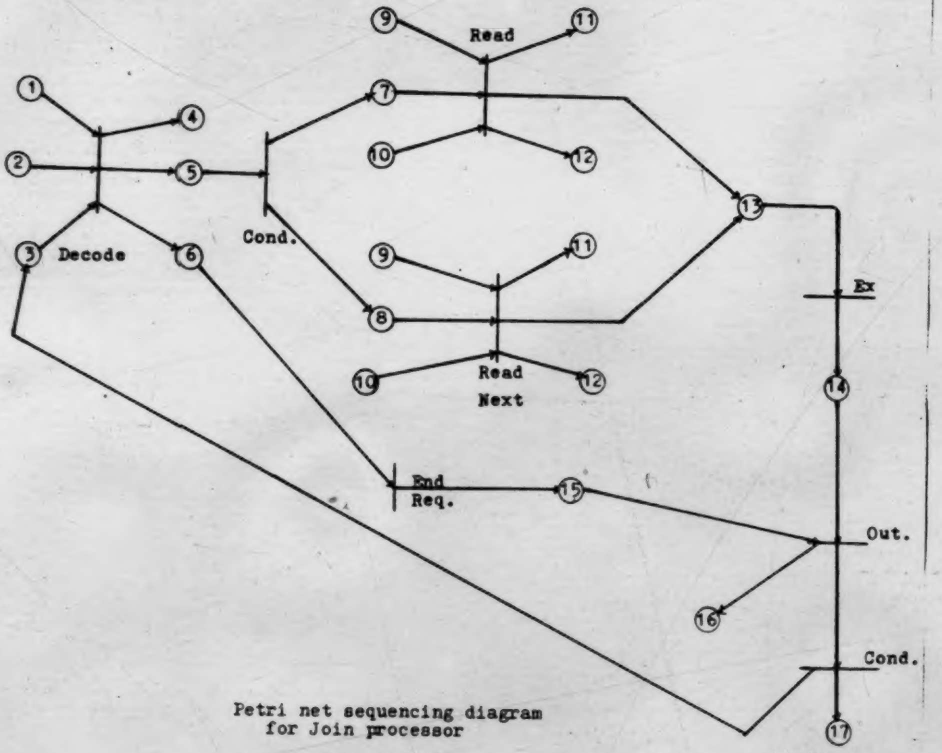
[5] OUTPUT :A control signal to be issued by the 'Timing Chain' circuit if it has received a conformation to the 'END-REQUEST' message. This signal will allow the transfer of the result.

[6] EXEC :A control signal to both buffers A and B to start performing the actual algorithm (to be explained under theory of operation Section).

The following are the condition places in Figure 3.5:

[1] A new operation packet has arrived at the processor and is ready to be decoded.
[2] The processor is free to accept executing an operation packet.
[3] The output destination bus(es) has (have) enough room in its (their) queues.
[4] The next operation packet could be accepted by this processor.
[5] All operands (pages) are ready to be read into buffers A and B.
[6] The decoding operation has been finished.
[7] Pages are coming from outside the processing station.
[8] Buffer B operand (page) is broadcast from another processor in the organization (other processor in the station).
[9] A page is ready to be read into buffer B.
[10] A page is ready to be read into buffer A.
[11] Buffer B is ready to read a new page.
[12] Buffer A is ready to read a new page.
[13] The hardware algorithm is ready to be activated.
[14] The result is ready to be output to its destination(s).
[15] A response to the 'END_REQUEST' message has arrived.
[16] The result has been output to its destination bus(es).

The petri net in Figure 3.5 should be interpreted as follows: Assume the arrival of an operation packet to perform a Join between two pages of relations T and R. The operation packet is first decoded by the decoder to find the relative positions of the θ operands ($T \infty_\theta R$) in which θ may be any of the operations (= <> > < >= <=) . It is important to note that more than one predicate at the same time can be handled using the ALAP. Then a 'READ' command from the 'Timing Chain' is issued after first checking the availability of both operand pages. After the loading of both pages at buffer A and B, the execution cycle is to be started. Each tuple from buffer B is moved to the mask register of buffer A (the ALAP) where the θ conditions are tested. The matched tuples are output one after the other to the concatenation algorithm, which will perform the actual join.


9
Read
11
1
7
4
10
12
2
5
13
3
Decode
6
Cond.
9
11
Ex
8
10
Read
Next
12
14
End
Req.
15
Out.
16
Cond.
17

Petri net sequencing diagram
for Join processor

Figure 3.5

During the processing of the tuples of buffer B, an 'END_REQUEST' message will be sent before the actual ending of processing in order to allow time to receive a replay if the output bus is free. Once the underlying processor has joined the assigned operation packet, the next command it is going to execute is 'READ-NEXT' instead of 'READ'. This can be seen from Figure 3.5 in which all the inner pages should be broadcast through the whole 'Ring' in order for the Join to be completed.

If the number of pages in the outer relation is greater than the number of available join processors in the processing station, then the whole process should be repeated for the rest of the outer relation pages. The number of processors assigned to the underlying operation depends on factors such as the size of source relations, the query priority and the system load.

There are several reasons behind using this nested-loop algorithm to perform Joins in the proposed machine. Although, it is well-known fact that the merge-sort algorithm is more efficient for the case of very large databases, it is not suitable for data-driven environments for the following reasons:

[1] The merge-sort join operation is an off-line operation; it requires the collection of all source relations for sorting before it produces any output. This absorption to the flow directly contradicts the concept of data-flow query processing strategy.

[2] Although the absorption of source relations would improve the Join productivity rate,afterwards, this may create a buffering bottleneck at the joins' output place(s).

[3] The requirement of the relations total availability at input ports follows the concepts of the 'packet level' query processing strategy. This requirement imposes a sort of sequential ordering on net operations which restricts the degree of allowed parallelism in data-driven environments and directly affects the performance of the strategy compared with un-restricted data-driven strategies.

In [BoD81] the pure data-flow query processing strategy was shown to superior over the packet level query processing strategy.

[4] In [ChS75] and [Yao79], the importunity of finding algorithms for relational database operations that support pipelining excludes the merge-sort techniques. It was shown that by using algorithms that support pipelining, one can utilize pipelining within primitive operations as well as for consecutive operations in the query net. If the processors executing primitive operations were fast enough, one thus can process the data streams coming in from disk without slowing them down.

[5] In [Gli83] the merge-sort algorithm was used in a pipelining environment with n merge processors and buffer memories for 2**n tuples. This had a linear time execution, consequently it was claimed that, by using this previous type of sort, the sort-merge join is suited for pipelined data-flow execution environments.

That may suggest for very large databases a sort of preprocessing to be done by the host processor in order to prepare chunks of appropriate sizes for the proposed machine.

Yet another possible solution for handling very large databases could be an extension of the architecture, as will be described in Chapter 6, which provides a separate ring of processing units to handle Join operations. In this type of 'Join ring' the availability of a large number of processors would allow the nested-loop join algorithm to perform much better than the sort-merge algorithm because of its higher sensitivity to the number of participating processors [VaG84].

3.5.2.2. Project processor station

Given a relation R as a source relation, a processor reduces it to a 'vertical' subrelation by discarding all domains other than the required domains. Since discarding may introduce duplicate tuples, the duplicates must be removed in order to produce a proper relation. The processors in the projection organization (station) are interconnected by a two-way uni-directional bus as shown in Figure 3.6. The removal of duplicates is done as follows:

[1] Each processor initially deletes its "intera-page" duplicates by copying the contents of its buffer A (the ALAP) into its buffer B, then matchs a tuple at a time from B with the page in A and deletes duplicates.

[2] Each processor, in turn, broadcasts its page through the upper uni-directional bus (see figure 3.6) and then exits (i.e, outputs its contents through the lower uni-directional bus).

[3] If processor P_j receives page i, then j>i. P_j compares the two pages and eliminates any duplicates found from its page. Note that P_j will never see page i if i>j (because of the uni-directional bus). Consequently it is guaranteed that only one copy of each tuple will remain in the relation (the copy will reside in the lowest numbered page of all the pages that had a copy of it).

[a] Functional block diagram

The functional block diagram is identical to the one in Figure 3.4, since it is a double-buffer processor. In our case the concatenation algorithm need not to be executed as long as the decoder detects a Projection not a Join operation. The functions of the Decomposer and Composer ,in figure 3.4, will be described in Chapter 5.

[b] Operation description

The Petri net in Figure 3.6 shows the command sequence to handle duplicate elimination within each processor. The following is a description of the commands and conditions in the net that have not been described in the Join net:

[1] SET : The 'Timing Chain' would issue this command in case buffer A (the ALAP) is empty after or during executing the sequence of eliminations (i.e, the whole tuples in buffer A page were duplicates). In this case nothing is to be output, so the SET command is used to indicate that the processor is free to be assigned a new operational packet.

[2] EXEC : This command executes steps [1] and [2] of the duplicate elimination procedure.

The following are the condition places found in Figure 3.6

[1] A new operation packet has arrived to the underlying processor and is ready to be decoded.

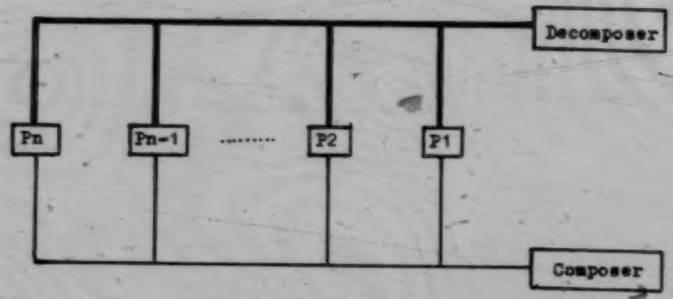
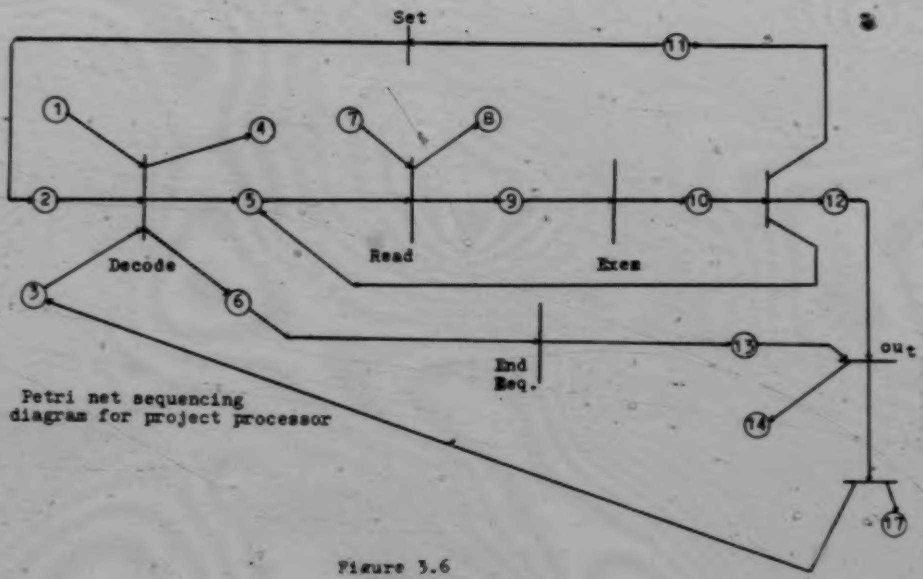

Decomposer
Pn
Pn-1
....
P2
P1
Composer

Figure 3.6:Project Processing Station


Set
11
1
4
7
8
2
5
9
10
12
Decode
Read
Exec
3
6
End
Req.
13
out
14
17

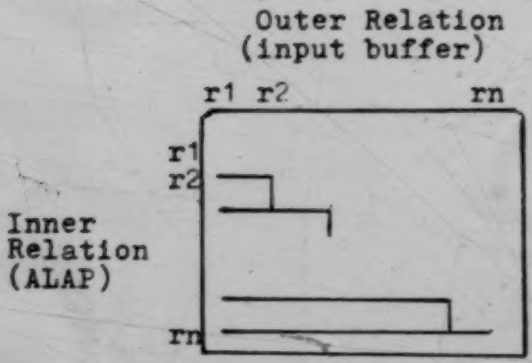
Petri net sequencing diagram for project processor

Figure 3.6

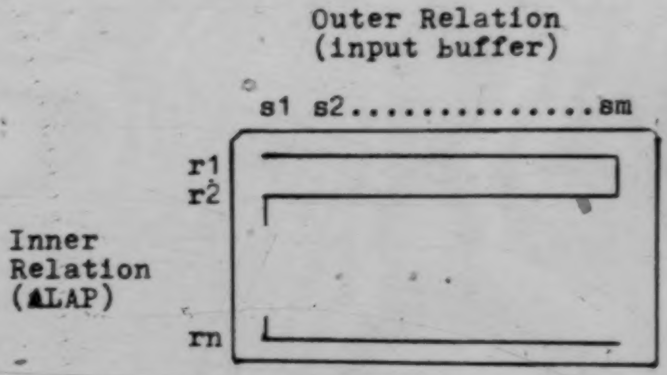
[2] The processor is free to accept executing an operational packet.
[3] The output destination bus(es) has (have) enough room in its (their) queues.
[4] The next operation packet could be accepted by this processor.
[5] All operands (pages) are ready to be read into buffers A and B.
[6] The decoding operation has been finished.
[7] A page is ready to be read into buffer A.
[8] Buffer A is ready to read a new page.
[9] The hardware algorithm is ready to be executed.
[10] Execution of the current page has been finished.
[11] Buffer A (the ALAP) is empty (no tuples to be output).
[12] The processor has broadcast its page and is ready to output its result.
[13] A respond to the 'END-REQUEST' message has arrived at this processor.
[14] The result has been output to its destination bus(es).
[17] The output destination bus(es) is (are) full.

For deleting duplicates, relation pages should be delivered to the duplicate elimination station according to the boolean matrix given in Figure 3.7. In the figure, the column label pages r_1, r_2, r_3 ....r_n represent pages to be broadcast from buffer B of the processor participating in the processing station. The row label pages (in the figure) are to be stored in buffer A and to be modified by the tuples in buffer B. The figure shows a schedule for a page-swapping strategy that achieves a minimum number of swapping counts [MKY81]. In the figure, assuming relation R has n pages, then the number of all pairs to be looked at is t=(n**2-n)/2

According to [MKY81] there is an optimal page swapping count of t-1 page swappings for scheduling the processing of t pairs of pages. The strategy presented in Figure 3.7 has the optimal number of page swapping counts. For the join operation, our inner-outer broadcasting technique has minimum number of page swapping counts. Since, the Boolean matrix of figure 3.8 shows that the schedule produces an optimal number of t-1 pages (t here is equal to (nm -1)).


Outer Relation
(input buffer)
r1
r2
rn
r1
r2
rn
Inner
Relation
(ALAP)

Page Pair Graph
for duplicate
elimination
Figure 3.7


Outer Relation
(input buffer)
s1
s2
sm
r1
r2
rn
Inner
Relation
(ALAP)

Page Pair Graph
for Join Operation

Figure 3.8

3.5.2.3. Update processors (Delete and Modify)

An important property that must be preserved when performing update operations is that no duplicates are to be introduced as a result of the update. One can, then, distinguish between two types of update qualification clauses: 'simple' and 'complex'. A 'simple' qualification is one that may be applied in a single scan to the relation. A 'complex' qualification is one which requires performing some inter-relation operations (like join) in order to determine the tuples to be updated.

3.5.2.3.1. Delete processor station

For the 'simple' qualification delete operations, the delete processor extracts the deletion criterion and loads it into the ALAP. Tuples satisfying the deletion criterion are removed from the page in the ALAP, and the rest of tuples are output.

[a]

Functional block diagram

Figure 3.9 shows the single buffer processor for handling simple delete operations. It consists of the same functional components as the double buffer processor except it has only one buffer which is the ALAP.

[b] Operation description

The petri net in Figure 3.6 can be used to describe the operation of the simple deletion processor, with one simple difference which concerns the interpretation of the EXEC command:

EXEC : Extract the deletion criterion from the deletion operation and load it into the ALAP. Then tuples satisfying the deletion criterion are disabled from being output by the ALAP, and the rest of the tuples are output.

The condition places have the same meanings as described in Figure 3.6.

For complex predicate delete operations, a modified Join processing station could be used which loads the tuples from the relation to be manipulated in buffer B and the other relation's pages in buffer A. In this modified Join version the EXEC command will delete the qualified tuples instead of materializing their join with the other relation matched tuples.

3.5.2.3.2. Modify processing station

One has to distinguish between two cases: whether the modified attribute does or does not contain the relation key or part of it, to be sure that the modify operation will never result in any duplicates. If the modified attribute does not contain the relation key or part of it, then the modification will never result in any duplicates. In this case, one can process the Modify using the 'Simple' delete processor to match the qualified records, but instead of deleting them, the appropriate attributes are modified.

On the other hand, in the second case ,where the modified attribute(s) may contain the relation key or part of it, one must check for duplications. The modification operation, in this case, works in three phases: (1) The first phase produces another copy of the relation ,call it (old), to be used in the case of backtracking if it discovers the existence of duplicates. (2) The second phase works on the relation by performing the 'Simple' modify approach described previously. (3) The third phase checks for duplicates using the same duplicate elimination approach, but in duplicate checking mode (instead of duplicate elimination mode). If there are any duplicates, it should delete the produced relation (new) and restore the (old) relation then report an error message.

[a] Functional block diagram:

The functional block diagram of the 'simple' modify is similar to the single buffer processor in Figure 3.9.

The petri net in Figure 3.6 could also be used to describe the simple Modification operation, with one simple difference, which concerns the interpretation of EXEC command:

EXEC : Extract the modification predicate from the modification operation and load its search criteria into the ALAP. Then tuples satisfying the search criteria are modified according to the modification predicate(s).

The condition places have the same meanings as described in Figure3.6.

3.5.2.4. Select processor

Selection is done using a single buffer processor, in which the relation pages are loaded into the ALAP, and the searching criteria is loaded into the mask register of the associative memory. Then each processor of those participating in the Selection station enables only those pages that qualify from the search predicate(s).

[a] Functional block diagram:

The functional block diagram is the same as the one in figure 3.9

The petri net in Figure 3.10 (the same as simple deletion) could be used for selection operations, with only one difference concerning the interpretation of EXEC command.

EXEC : Extract the search predicates from the operation packet and load them into the ALAP's mask register. Then only qualified tuples are output.

3.6. Data-flow /Mixed-flow architectures

In the previous sections a number of specialized functional processors for all relational database operations have been proposed to be incorporated by the

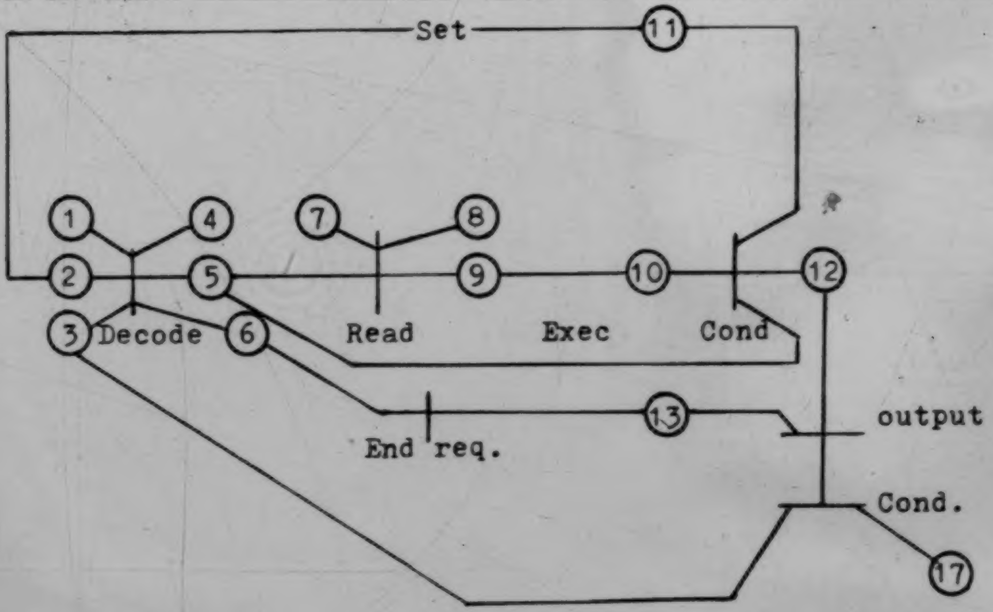


Figure 3.10 Petri net sequencing diagram for Select

architecture. A data-flow query processing strategy which provides any number of those processors to execute each transition in the query net has been studied. Furthermore, it has been shown that it is required to dynamically adjust which processors are executing which transitions in the query net, in order to distribute processors across all transitions and maximize performance.

In this section, our major objective is to describe an architecture which implements the previous scheme of utilizing resource pools of the specialized functional processors and buffers to physically map query structures onto the hardware structure, taking into consideration the architecture requirements mentioned in section 3.4. That architecture should also be able to provide a 'reasonable' solution to the question of how the elementary transactions in a query net could be assigned to functional processors in the machine in a manner which allows the three things mentioned in Section 3.3 to occur:

[1] No concurrency should be lost, if at all possible.

[2] The solution should use a minimum number of processors.

[3] The communication overhead should be reduced as much as possible.

Chapter 4 will apply algorithms to direct the data-driven nature of the pure data-flow query processing strategy to satisfy the latter three goals, (as much as possible) but these will be used in later stages of the design. Our intention now is to describe an architecture for the first set of requirements.

The organization of the remainder of this Chapter is as follows: Section 3.6.1 first characterizes an architecture type called data-flow architecture, then shows that its characteristics match, to some extent, our set of requirements. Then due to some shortcomings in the nature of data-flow it deviates to a 'mixed-flow' architecture type. Then Section 3.6.2 shows that some 'mixed-flow' model can totally match our set of requirements. In addition it allows a greater chance for the second set of optimization requirements to be achieved.

3.6.1. Data-flow architectures

Data-flow computer architectures are a newer class of architectures [TBH82] which utilize multiprocessors and are based on a data-driven computation model called a data-flow computation model. The data-flow computing model [Haz82] [TBH82] is derived from program representation, not from the computer architecture as it is the case in the Von Neumann model. In the data-flow model there is no control flow; instead, the flow of data guides the computation. Thus, each operation in the computation is executed when and only when its input operands are available. A data-flow program can be mapped to a directed computation graph, where nodes represent operations to be performed and arcs represent data paths between operations.

A node (operation) acts (fires) only when its input tokens (operands or data) are available. When an operation is fired, all tokens are moved from its input arcs to its output places through its outgoing arcs (data paths). A node may have any number of input and output arcs (input parameters and copies of output values). Arcs in the graph operate as queues, delivering one token (data value) at a time to the node (operation). Data (tokens) flow forward in the computation graph and each node fires when its operands are available.

The goals of data-flow architectures as seen by researchers of the field are:

[1] Increasing computer performance through concurrency.

[2] Direct support of functional programming languages resulting in increasing reliability due to easier verification of functional programs.

[3] Exploit VLSI designs, through a computer organization consisting of identical complex functional units connected together in a regular structure with little off-chip communications.

Basically, data-flow systems differ from control-flow (Von Neumann systems) systems in a number of characteristics [Haz82]:

[1] The built-in synchronization of process execution in the data-flow system is automatically enforced by the dependency of operation executions on availability of input operands. In the control-flow model synchronization among concurrent

processes must be explicitly provided using semaphores or similar mechanisms.

[2] The most visible difference between the two models is the execution ordering. In the control-flow there is a very deterministic total ordering at compile time, and execution at run-time merely follows this ordering. In the data-flow model, there is an implicit partial ordering on the basis of data dependencies before run-time. At run-time, this partial ordering, together with other constraints such as resource availability determines the total ordering of execution.

[3] Another visible difference is the time of resource allocation for operations. In the data-flow model, the allocation is delayed until the operation has its input operands available and is about to execute and then the resources are no longer needed once the value of the result of the operation is available to other operations. In contrast, in control-flow systems, resources may be allocated for an operation long before it is executed and some resources may remain allocated for the entire duration of program execution.

It is argued that the previous characteristics match our set of architecture requirements. The first requirement of the possible mapping of current demand to the system onto a hardware structure is possible, since in data-flow architectures, resource allocation processes are delayed until the operation has its input operands available and is about to execute. That means, only the needed resources are allocated only when they are demanded, so the system could be considered self-adjusting, in the sense that only a number of processors exactly equal to the current demand without any wasted 'fragments' are utilized. That also provides the flexibility of dynamically adjusting which processors are executing which transitions in the Query net.

Furthermore the requirement of easily imposing concurrency control mechanisms on the architecture is well matched by the data-flow architecture property of determining total execution ordering using data dependencies besides external factors. Concurrent control mechanisms could be among those external factors, that means they could participate in determining the total order of execution.

In the data-flow model, if one of two concurrent recurrences takes longer to compute than the other, computation of the latter one need not be slowed down, since each newly created instance of a result can be individualized by attaching to it an activity number. In essence, this fills the queues on the arcs (data paths) in the computation graph. This look-ahead property increases parallelism in a manner similar to the 'Dynamic concurrency' concept which is required by the data-flow query processing strategy.

In addition to the previous characteristics which meet our set of requirements, the interconnection methodology used by data-flow architectures which establishes logical interconnections among processors and between processors and the intermediate buffers, instead of physical ones (like the crossbar switch used in DIRECT), is of great importance in our mapping of query structures onto the hardware structure. This interconnection methodology (not explained here, but described in more details in Chapter 5) is able to support MIMD environments without being a bottleneck in the design.

3.6.2. Mixed-Flow architectures

The results of the design considerations presented in Section 3.4 showed that our architecture is likely to be a multiprocessor system with asynchronous communication and distributed control. The previously described architecture meets these considerations except the last one (distributed control), since data-flow architectures don't have any control. Beside that, there are number of problems in using pure data-flow architectures in processing queries:

[1] It is not possible to satisfy the functional requirements of complex database queries in a pure data-flow architecture. The difficulties in operating upon the large data structures handled by such queries, are reported in [Gajski82].

[2] [Gajski82] pointed out that there is a conflict in using broadcasting on the one hand (which introduces artificial synchronization into the execution of queries but reduces the amount of input/output), and data-flow (which ideally should be entirely asynchronous) on the other hand. For example, the broadcasting of pages to processors of a Join processing station needs a way to control the arrival of pages from previous operations. This to allow the broadcasting to be done in a well-defined pipeline fashion (as in the matrices of figures 3.7 & 3.8) instead of in a purely data-flow fashion.

[3] According to the diverse performance capabilities of the specialized components, one should expect to have intermediate queues of pages created from these performance divergences. A pure data-flow architecture does not provide these kinds of intermediate bufferings.

[4] Processing queries in a pure data-flow architecture has a very low capability in performing decision-based operations, in this sense; at times decisions need to be made based on insufficient information due to the pipelining nature of the strategy. For example, when a Join is initiated there is no way of knowing the sizes of intermediate relations so as to decide which relation should be the outer relation.

Because of the nature of the previous shortcomings, a 'mixed-flow' architecture that lies in between data-flow and control-flow architectures could be useful in overcoming most of these shortcomings.

In fact, the possibilities between strict control , on the one hand and strict data-flow on the other hand, were investigated in [Haz82]. Based on the three basic differences, mentioned in Section 3.6.1, between control-flow and data-flow architectures, Hazara assumed that there are only two possible values (yes/no) for each of the above, so one could have six possible distinct computation models representing 'mixed-flow' architectures. This division allows us to have ,for example, an architecture of 'mixed-flow', which has the built-in synchronization of data-flow, but has its operations ordered for execution before run time , as in the pure control-flow machines. This means one could build a 'mixed-flow' machine which is exactly tailored to the architecture requirements sought.

CHAPTER 4

Mixed-flow query processing strategy

4.1. Introduction

This chapter looks at the computation space model, a model used to depict a mixed-flow query processing strategy serving in the proposed design. The underlying belief behind introducing this new query processing strategy is that it will alleviate the shortcomings of the extreme data-flow query processing strategy mentioned in Chapter 3; besides, it is obligatory that a Computation model for a mixed-flow machine itself should be carried out in a mixed-flow fashion.

In Section 4.2 of this chapter, the basic query net, mentioned in Chapter 3, is shown to be superior to dependency graphs as a tool for query representation. Then Section 4.3 presents a number of extensions to the basic query net. These extensions will serve in defining the general query net (or called computation space net), and facilitate the representation of multi-query environments. Section 4.4 first defines our mixed-flow query processing strategy; then uses both the basic query net and the general query net for examining the two query processing strategies: pure data-flow and mixed-flow. Then a simulation that has been used to evaluate them is described in Section 4.4.2.

4.2. Basic query net

Section 3.3 presented a formal description of the basic query net, and claimed its superiority over, the older, dependency graphs as a tool for query representation. This section justifies our claim, and shows how the model offers a number of facilities which serve our purpose.

The petri net model was adapted to model database transactions, instead of dependency graphs, in the framework of information systems design (see for example [Rol82] and [LeH82]). It was shown that besides its capabilities to describe the dynamics of transactions, it is also convenient for the analysis of database properties such as critical sections and deadlocks. For example, (1) A petri net is live if there always exist a firing sequence to fire each transition in the net. By proving the liveness property of the net, the system is guaranteed to be deadlock free, and (2) A petri net is bounded if, for each place in the net, there exists an upper bound to the number of tokens that can be there simultaneously. By proving the boundedness property of the net, the number of buffers required between asynchronous processes can be determined and, therefore, information losses due to buffer overflow can be avoided. Moreover, (3)a petri net is properly terminating if it always terminates in a well defined manner (no tokens are left in the net). By verifying the proper termination property, the system is guaranteed to function in a well-behaved manner without any side effects on the next iteration.

Despite these advantages, there are two main problems with using petri nets in transaction representations [OzW82]. The first one is the lack of time concept which results in instantaneous transaction firing. This restriction could be addressed by using 'Timed Petri Nets' [RGD82]. The second deficiency is the absence of a mechanism whereby tokens which represent jobs in the system can collect data.

These deficiencies, and, in addition, the inability to specify an upper bound on the number of tokens at each place, have led to the development of the query net model mentioned in Chapter 3. In addition to the capability of the query net model to overcome the previous problems, it also offers a number of facilities which serve our purpose:

[1] It allows page reentrants through the net, by associating an iteration number with every new invocation to the net. This facility allows us to model the 'dynamic concurrency' concept mentioned in Section 3.6.1.

[2] The model is so close to the proposed architecture that the mapping of its elementary transactions into machine instructions is a straightforward process (as will be shown in the next Chapter).

[3] The model facilitates the study of some machine performance measures: finding the optimal buffering distribution on a specific query net, and finding the optimal process execution scheduling.

4.3. General query net (Computation space net)

Section 3.3 gave an example of a basic query net and showed the steps for deducing it. In fact the query net in Figure 3.1 is called a 'Stream-oriented' query net, because it has no conditions (choices), no selectivity at any place, and no iterations. This section will define what is to be called a 'general query net' (or Computation space net) ,by introducing some extensions to the basic query net, for two main reasons: first, to facilitate the representation of multi-query environments, and second, to use it in describing our mixed-flow query processing strategy.

[Def.4.1]

Places represent intermediate buffers in which queues of pages are stored for transactions. Each place, P_i, is defined as a tuple $[M(P_i), T(P_i)]$,where $M(P_i)$ represents the place buffering capacity,i.e, the maximum number of pages that could be stored by the buffering facility at the place P_i. $T(P_i)$ is the place type. Places could be classified as Fork, Join, or Unity places. P_i is a 'Fork place' if the set $g \bigcap \{\{P_i\} \times A\}$ (see Section 3.3 for variable definitions) has more than one element; it is called a 'Join place' if the set $h \bigcap \{ B \times \{P_i\}\}$ has more than one element. Finally, it is called a 'Unity place' if none of the sets g $\bigcap$ $\{\{P_i\}$ X A$\}$ and h $\bigcap$ $\{$ B X $\{P_i\}\}$ have more than one element.

According to the routing mechanism, a place ,P_i, could be classified as a selective or nonselective place. Moreover, it is said to implement either a Fixed or Functional selection according to the nature of the routing mechanism. For example, a Fork place ,P_i, is said to implement a non-selective routing if and only if, for all in-directed pages g_i to P_i, an out-directed page g_j appears on each of the out-directed arcs from P_i. A Fork place ,P_i, is said to implement a selective routing if there exists a routing criterion which establishes, for each in-directed page g_i, subsets of the out-directed arcs on which g_j will appear. Finally, Fixed and Functional selection are defined. For example, a selective Fork place, P_i , is said to be Fixed selective if the routing criterion is fixed, whereas it is said to be Functional selective, if the routing criterion exists in the form of a functional relation.

The reason for establishing these classifications is to have a means by which one can represent the updates to the Computation net. Updates mean improving the processing power at certain transitions in the net, or increasing the buffering facility at certain places. For example, to improve the response time of a certain query, one can improve the processing power at some of the elementary transitions on this net's critical path(s). Doing this, one may have to increase the sizes of the intermediate buffers on these paths. By having the notions of Fork and Join and selectivity in the general query net, updates can be easily represented (an example can be found in Section 5.2.1).

[Def.4.2]

Transitions represent subqueries. Each transition ,t_i,is defined as a 4-tuple [S(t_i), T(t_i), Y(t_i), F(t_i)] where S(t_i) is the transaction state. During page propagation through the net (query execution), a transition t_i may be in one of the following possible states:

[1] Waiting : a transition ,t_i, is waiting for its predecessors to be fired (more precisely, for its operands to be available). Initially, all transitions are in the waiting state except those at the entrances of the net.

[2] Current : a transition t_i is in the current state when all of its input requirements are satisfied (available in the buffers); that is, all its preceding transitions have already produced results.

[3] Firing : a transition is fired for execution.

[4] Completed: execution of the transition is completed and the results either are moved to the output buffers for another transition or produced as final net output.

T(t_i) is the duity assigned to this transaction (the work-load it has to finish). The use of that field will be apparent when the mixed-flow query processing strategy will be described.

Y(t_i) is the transition time;i.e, it is the time required by the transition to move tokens from the selected input places to the selected output places.

F(t_i) is the transition function. Each transition is associated with a transaction. Transition functions could be any of the relational database primitive operations.

[Def.4.3]

A token is a page with n tuples defined as K[1..n] and the ith tuple referred to as K[i]. A token can be in one of three states while it is propagating through the net:

[1] Reserved : during the firing state of a transition ti, this token was residing at a place $P_i \in \text{In}(t_i)$.

[2] Traveling : during the firing state of the emitter transition ,t_j, (a transition that emitted that token), till the arrival of that token to its destination place, that

token is said to be in traveling state.

[3] Available : otherwise.

Also associated with each token three fields:

[a] Destination places: addresses of destination places that the token is supposed to be delivered to.

[b] Iteration number: an incremental integer used to distinguish different net invocations.

[c] Source name : the name of source transition from which this token is emitted.

These definitions facilitated the manipulation of pages through the net, especially in our simulation programs, where one has to distinguish between the times of pages being available and being ready for firing.

[Def.4.4]

An inhibitory arc is represented by a dotted arc, and goes from place to transition in the net. It effects the transition firing in the following manner: "A transition ,t, is enabled if and only if each entry place, P, connected to t by an ordinary arc contains a token (a page), and each entry place connected to t by an inhibitory arc is empty."

The inhibitory arc was introduced in the model to facilitate the representation of priorities, which aid in solving problems of mutual exclusion and deadlocks.

[Def.4.5]

A transition which selectively directs the tokens according to some selection criterion is represented by a positive '+' sign and is called a choice transition.

The general query net provides us with two main facilities over the basic query net: (1) The incorporation of choices and conditions allows us to represent a computation net for a multi-query environment which is to be served by our machine.

(2) The incorporation of inhibitory arcs gives us the tools to impose concurrency control mechanisms on the query nets.

Figure 4.1 shows an example for a computation space net of a multi-query environment of an Airline reservation system. This example is limited to only four activities in the environment: Ticket reservation, Ticket refunds, Ticket purchasing and Ticket Okay. System relations and elementary transactions explanation are shown in Appendix A.

The Computation space net, shown in the figure, is a collection of query nets for different environment functions (activities). The net was drawn by first deducing a query net for each activity individually. Then according to I/O dependencies, integrity constraints, and concurrency control rules, extra inhibitory arcs and locks are imposed on different parts of the Computation space net. Figure 4.2 shows how concurrency control mechanisms are imposed on the Computation space net. In this figure, two critical sections are taken care of by inhibitory arcs.

[1] A critical section between the Ticket Okay activity and Ticket purchasing activity in sections R71, R21 is being controlled by a priority arc introduced to enforce consistency. Moreover, a lock is introduced around critical section 1 to enforce an integrity constraint: it is not allowed to issue more than one Ticket Okay without checking the plane capacity.

[2] Another priority arc has been inserted between the two activities of Ticket reservation and refunds, as shown in Figure 4.2. The semantics of this corresponds to the fact dhat it might be the case that there is no more space on a flight number A, while there is a request for a reservation on it; at the same time, another passenger is refunding a Ticket for that flight. Similar to (1) ,another lock has been established around 'reservation' activity to enforce the integrity constraint, where it is not allowed to issue more than one reservation without

checking the plane capacity.

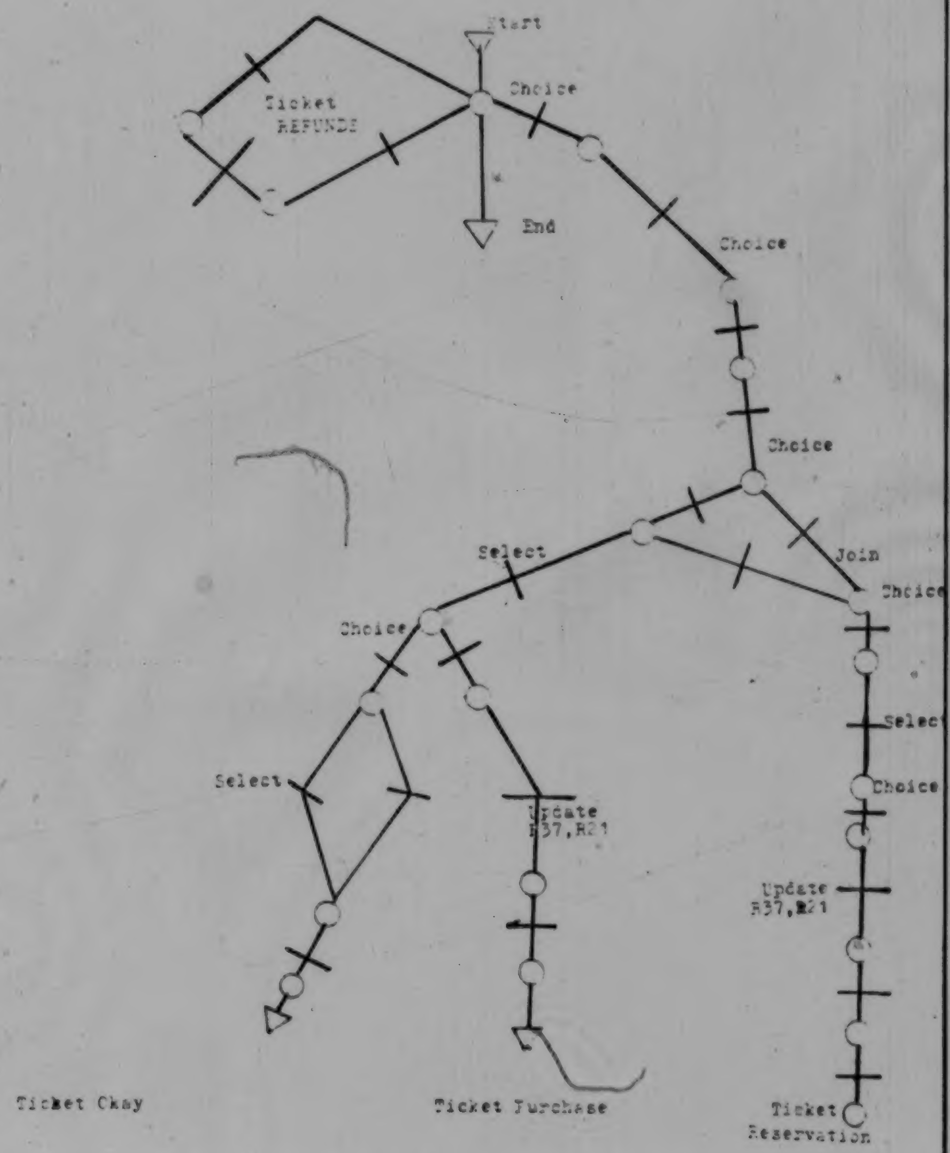
This section was meant to demonstrate that the Computation space net (such as the one in Figure 4.2) can provide a complete description of the interrelations among the major system activities. It could be viewed as an intermediate language to describe explicitly the maximum parallelism that exists in the target system. This would allow us to examine different firing sequences (process scheduling strategies) to handle the operations in the multi-query environment without running into concurrency problems. The selected firing sequence is then to be translated into an executable concurrent/pipelined scheme on a target system. As will be shown in the next chapter, the mapping of the executable scheme onto the target system is to be done by the 'Resource Allocation Unit', and managed by the 'Process Scheduling Unit'.

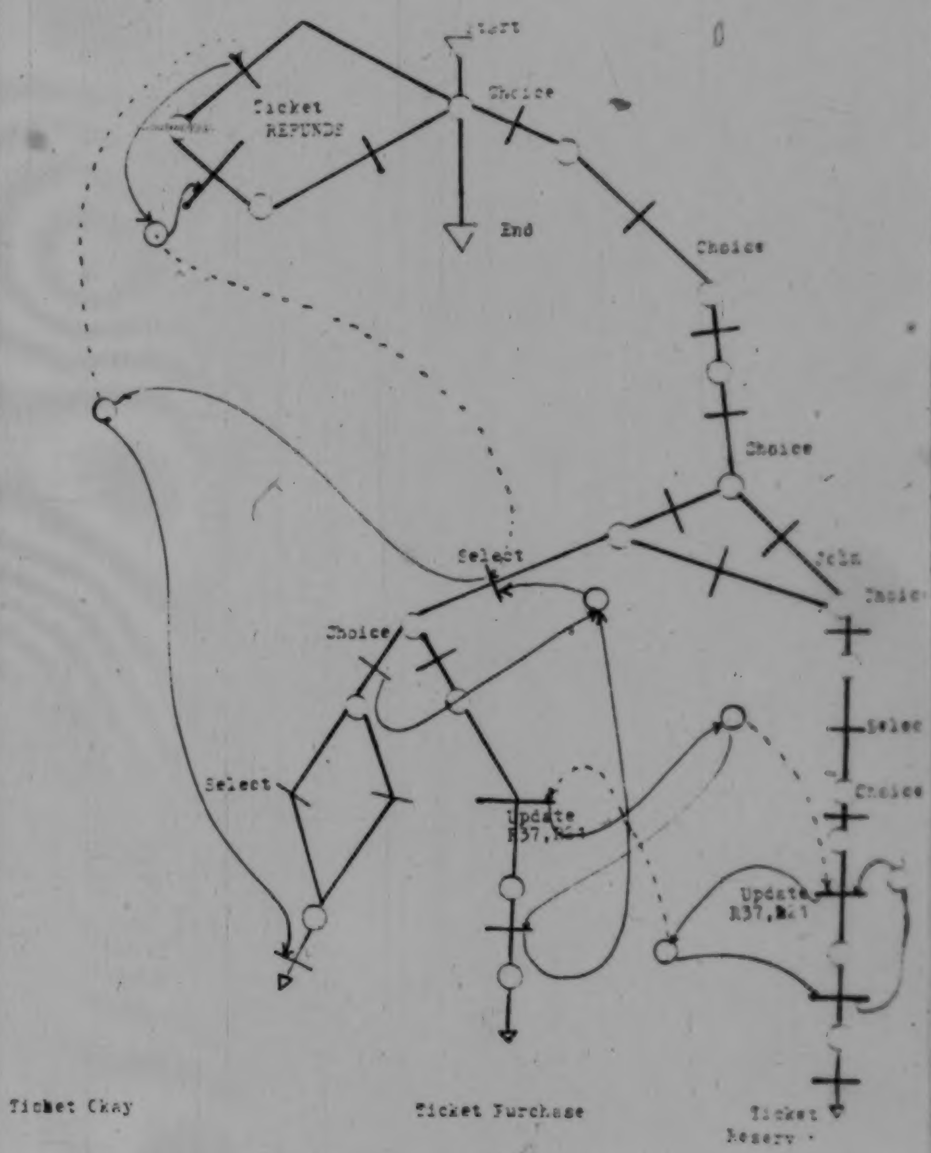
4.4. Selection of processing strategy

In Chapter 3, the basic query net model was used to describe the data-flow query processing strategy, and a number of advantages and disadvantages of this strategy were presented. This section will use the general query net to describe a mixed-flow query processing strategy. Then it will describe a simulation that has been used to evaluate different possible mixed-flow strategies (including the pure data-flow as a special case) and its results.

4.4.1. Mixed-flow query processing strategy

The Computation Space net represents several activities in a multi-query environment. One possible scheduling to serve this type of environment is the 'Time sharing' process scheduling. In this strategy, according to the query priority, size of source relations used, and system load, certain slices of time are assigned to sets of activities. Since each activity is simply represented by a query net, this ends up with a query net and a specific amount of time assigned to it. Our main concern in the


Start
Choice
Ticket REFUNDS
End
Choice
Choice
Select
Join
Choice
Choice
Select
Update R37,R21
Select
Choice
Update R37,R21
Ticket Okay
Ticket Purchase
Ticket Reservation


Start
Choice
Ticket REFUNDS
End
Choice
Choice
Select
Join
Choice
Choice
Select
Update R37,R21
Select
Choice
Update R37,R21
Ticket Okay
Ticket Purchase
Ticket Reserv

mixed-flow strategy is to schedule processing within every activity's query net, so that an improvement to the mean response time over all activities can be realized. This point can be revised a bit by first answering the question of what is the best progress that can be achieved in a set of active queries within the time slice assigned to it.

Our way of attacking this problem is to transform the problem of finding the optimal processing strategy for the query net in a specific time slice , into a problem of maximum flow/minimum cut. The reduction is reasonable since both problems are of the optimization type and the transformation is mainly from a time frame into a capacity frame (as will be shown). So the optimal schedule produced will assign to every transition in the query net a specific amount of work to do (this must be within its capability). Processing of that distributed work load is done in a data-driven manner (normally as in the pure data-flow query processing strategy). So the suggested strategy has a global control (assigning the work load for a time slice) and a data-flow nature within the time-slice.

The underlying rationale behind this strategy is: usually in a multi-query environment there is a huge demand and few resources to cope with it, and proceeding in a purely data-driven manner in such environments can cause very poor system performance. Figure 4.3 shows a simple example of a case where there is a select elementary transaction, followed by a Join elementary transaction. Suppose that within a time slice of T secs, the Join processing station is able to produce only 10 pages (assuming some selectivity factor and some number of processors participating in this Join station), whereas the select can produce a 1000 pages in the same time. Furthermore, assuming that the Join has utilized only 100 pages of those produced by the select to do its task, then 900 pages produced by the selection did not contribute to the progress (the 10 pages produced by the Join). On the other hand, in Figure 4.3(b), the application of the maximum flow/ minimum cut transformation determined

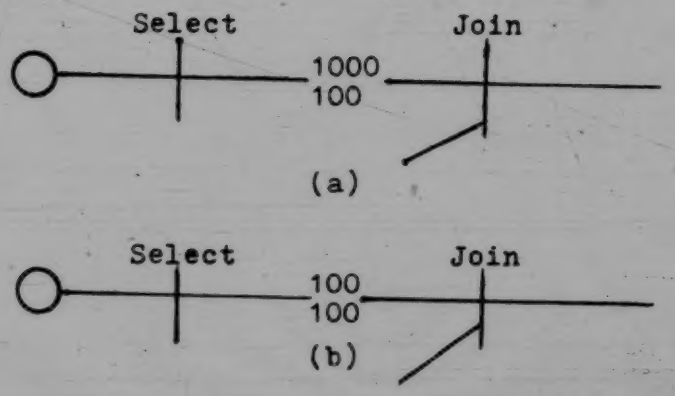


Figure 4.3: Controlled and uncontrolled flow

the exact number of pages needed to contribute to the progress (the 10 pages), and forced the selection to produce only 100 pages. This has saved all the selection processors and buffers participating in this select processing station a portion of T secs. These saved resources can be utilized in any other query in the environment which is being executed at the same time. Furthermore, this may result in starting an idle query, which was outside the environment due to lack of resources at a certain point in time.

Besides improving the mean response time, global control is also beneficial in preventing the occurrence of enormous queues in front of any compound processing station which is next to a fast one (as in the case in Figure 4.3). Another benefit is that the application of the maximum flow also has an important consequence in leaving the net properly terminating in a well-defined manner (no tokens are left in the net after the time slice). It thus assures that the process is guaranteed to function in a well-behaved manner without any side effects on the next invocation (next time slice).

In addition to the above advantages, the shortcomings mentioned in Section 3.6.2, which hampered the pure data-flow query processing strategy, could easily be rectified using this mixed-flow query processing strategy as follows:

[1] The conflict, mentioned in Section 3.6.2, between broadcasting on the one hand (which introduces artificial synchronization into the execution of transactions but reduces the amount of I/O), and the pure data-flow (which ideally should be entirely asynchronous) on the other hand, could be resolved by using the mixed-flow strategy. Global work load assignment could assign the required work loads according to the well-defined matrices defined by the broadcasting techniques that were proposed by Merrett (see the work of [MKY81]). This would provide us with a local data-driven pipeline that follows the broadcasting requirements.

[2] According to the mixed-flow query processing strategy, a controlled number of intermediate buffers are provided; this intermediate buffering is required because of the diversity in performance capabilities among the specialized relational database processors (described in Chapter 3) but was not supported by the pure data-flow query processing strategy (which supports no intermediate buffering in its ideal case).

[3] Finally, the capability of performing decision-based operations is expected to be improved by using the mixed-flow strategy since more information (due to the

global work load assignment) will be available at the times these decisions need to be made.

The following sections show first how to apply the maximum-flow/minimum-cut theorem to our optimization problem. Then they present a simulation done to compare the performance of the mixed-flow and the data-flow query processing strategies.

4.4.1.1. Reduction to maximum-flow/minimum-cut problem

In order to apply the maximum-flow/minimum-cut theorem to our query net , the following sets up some definitions.

[Def.4.6]

The capacity of a place ,P_i, denoted by C(P_i), is defined as the maximum number of pages that could be available at that place at any time within the given time slice, T secs. This value could be calculated knowing the execution times of all net transitions ,having in-directed arcs into that place (There is only one, since Query nets are defined to be Free-Choice Petri nets), using the following equations:

For entry places:

C(P_i) = Size of the source relation at the entry place.

For places other than entry places:

C(P_i) = T / time of producing a page at (t_i) (Sec/page). for all $P_i \epsilon$OUT(t_i)

On the other hand, a flow, f, at place P_i, denoted f(P_i), is defined as the number of pages, calculated by the maximum flow procedure, which are assigned to be fired (executed) out of that place.

[Def.4.7]

Also t is defined as an imaginary transition attached at the end of the active query nets to represent the destination (sink).

[Def.4.8]

A legal flow, f, through the query net Q, associates with each place $P_i \epsilon P$,an integer, $f(P_i)$, satisfying these conditions:

[1] $0 \leq f(P_i) \leq C(P_i)$ for each place $P_i \epsilon P$

[2] All C(k) are equal for $k \epsilon Out(t_i)$

[3] For single input transitions t_i (Single buffer processors)

selectivity factor of $t_i * f(P_i) \leq f(P_j)$

for all $P_i \epsilon In(t_i)$ & $P_j \epsilon Out(t_i)$

[4] For double input transitions t_i (double buffer processors)

selectivity factor of t_i * Page size * $f(P_i) * f(P_j) \leq f(P_k)$

for all P_i, $P_j \epsilon In(t_i)$ & $P_k \epsilon Out(t_i)$

Note that Selectivity factors are statistical figures associated with every transition in the net to represent the processing capabilities (production and consumption) at these transitions.

Informally, it is to say that f is a legal flow through the Computation net in a certain time slice, T secs, if it satisfies the four previous conditions. Condition (1) tells us that the number of pages that are allowed to be fired (for execution) must be non-negative and not exceed the capacity at that place (i.e, the maximum number of pages that could exist at that place during the time slice). Condition (2) states that the output pages from a transition should appear at all output places of it(this is from the nature of the net). Condition (3) states that any flow to the destination must have the property that that flow is conserved according to equation (3) at every single input transition in the net. Equation (3) states that at most a fraction equal to the selectivity factor of all incoming pages should appear in the outgoing places of ti. Condition (4) states the conservation law in cases of double input transitions;i.e, that no more than the selectivity factor of the eventual maximum number of pages produced out of the double input operation should appear in the outgoing places

[Def.4.9]

A transition, t_k, incident upon P_i and P_j in Q, is said to be useful from place P_i to place P_j with respect to a legal flow function f, if

$f(P_i) < C(P_i)$

and $f(P_j) < C(P_j)$

[Def.4.10]

The value of any legal flow f is defined to be

$V(f) = \text{Sum } f(P_i)$

for all $P_i \in \text{In}(t)$ where t is the sink.

[Def.4.11]

A path ,L,from place P_i to P_j in a net Q, is defined as a sequence of distinct places and transitions $L = P_i\ t_i$ ....P_j such that $P_k \in L$ (for each k=i,...,j) and is defined to be acyclic.

[Def.4.12]

A flow-augmenting path, with respect to a legal flow function f on a net Q is defined as a path L with the property that each transition $t_i \in L$ is useful from place $P_i \in L$ to place $P_{i+1} \in L$. A flow augmenting path from a source to the sink is called an s-t flow-augmenting path. It will be shown in the proposed procedure for finding the maximum flow, that an s-t flow-augmenting path is a path along which f can be augmented (i.e, the value of f can be increased).

4.4.1.1 Reduction to maximum-flow/minimum-cut problem

The study of the maximum network flow problem has been based on a fundamental theorem, first proven by Ford and Fulkerson in the mid 1950's. A variant of the theorem is applicable to our version of net maximum flow. The statment of the theorem and its proof are given in Appendix B.

Table 1 shows the figures used in the transformation from time domain to a capacity domain using a time slice of 24 msec. and page size of 10 tuples.

The transformation from time domain into capacity domain is shown by the example in Figure 4.4. Figure 4.5 shows the legal flow that is assigned by the pure data-flow query processing strategy. It is clear that the maximum processing power at each transition in the net is being utilized. On the other hand, Figure 4.6 shows a legal flow, fo, defined on the net. The first number represents the capacity of the place, whereas the second number represents the flow from that place.

Let us now consider a legal flow function defined on net Q whose value is maximum over the set of values of all legal flow functions defined on Q. Such a flow function is said to have the maximum amount of progress that could be reached in the net within the given time slice. Notice that the existence of a maximum flow function on a net is not open to question, since the net is composed of only finite capacity places (because of the finite time slice and the finite net transition execution times).

The problem of finding the maximum flow through the net is a variation of the known maximum flow/minimum cut problem with a different set of constraint equations. In the following a simple maximum flow procedure to be used to solve our query net maximum flow problem is described. The procedure is based on successively augmenting an existing legal flow function in each time slice period assigned to the net. The procedure begins by determining the amount of flow that is required to be popped from different net places in order to achieve a progress of one page at the sink. This amount is called the popping factor, PF, and is calculated by the backward labelling equations mentioned in the Appendix B proof. Then a flow potential value is calculated for each place in the net by the equation:

$$mp_i = Int\left(\frac{(C(P_i) - f(P_i))}{PF(P_i)}\right)$$

Notice: the notations used here are defined in Appendix B.

Then each flow augmentation is performed in three steps:

[1] First a place, Pr, with minimum non-zero flow potential over all net places is selected as the bottleneck of the net.

[2] mp_r units of flow are popped to the sink t through the net from every entry place reachable from Pr. The popping is done following the conservation equations mentioned in Definition 4.8.

[3] Finally the procedure updates the flow potential of each transition in the net through which the flow has been pushed, closing the transitions which become either saturated (unuseful), or unreachable from entry places or t. This prepares for the next flow augmentation to be assigned to the net.

Table 4.1 transformation table

Figures for the transformation.				
Transition	Exec. time	C	Selectivity	St*Ps
t1	.1m secs.	240	.01	.1
t2	.1m secs.	240	.01	.1
t3	.8m secs.	30	.001	.01
t4	.8m secs.	30	.001	.01
t5	.8m secs.	30	.001	.01
t6	.6m secs.	40	.1	1

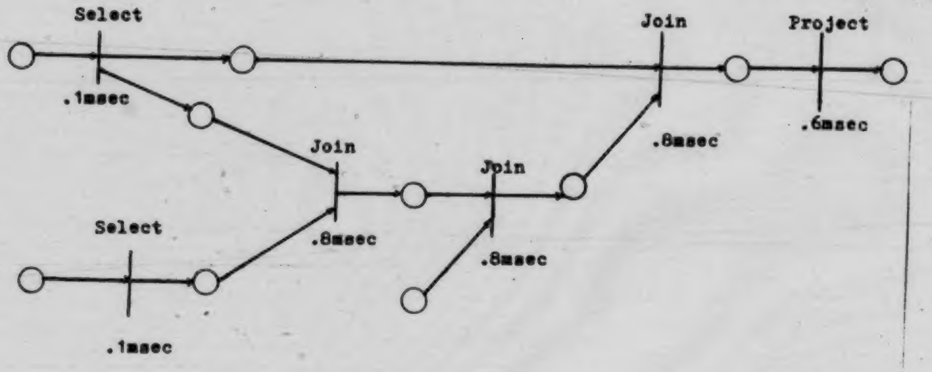
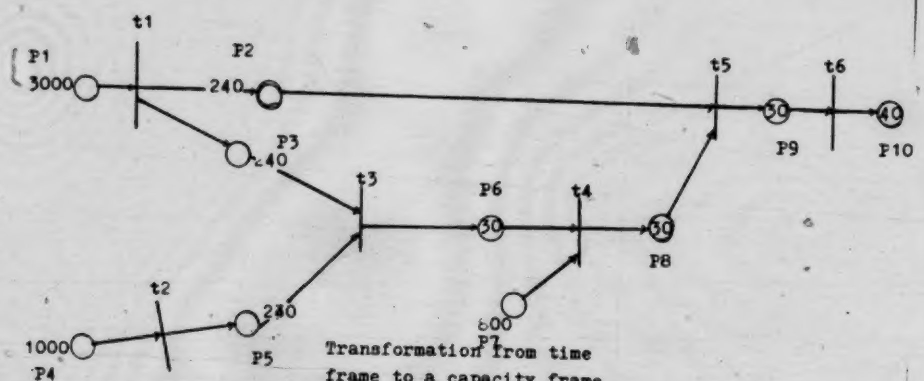

Select
Join
Project
.1msec
.6msec
.8msec
Join
Join
.8msec
Select
.8msec
.8msec
.1msec

Figure 4.4 (a)


t1
P1
3000
P2
240
t5
t6
P3
240
30
40
P9
P10
t3
P6
30
t4
30
P8
t2
240
600
P7
1000
P4
P5

Transformation from time frame to a capacity frame

Figure 4.4 (b)

Figure 4.7 shows, by an example, how to apply the procedure to a Query net. But notice, in the case of binary operations, the option of forcing one critical path to appear; this helps in providing the chance for the transitions on the non-critical paths to have a very low load. So their processors can participate for a very small portion of T secs (the time slice). Moreover, the procedure can alternate in choosing the critical paths from one time slice to the next time slice. This provides the opportunity for keeping the least number of processors busy in executing the query, and the non-saturated processors only participate at the beginning to finish their load portion; they then quit.

The figures in Figure 4.7 show the flexibility provided by this strategy in alternating critical paths.

It important to mention that the result of applying this strategy agrees with the recommended streaming which was suggested by Boral and Dewitt [BoD81].

4.4.2. Simulation results

In [BoD81] a comparison between four multiprocessor query processing strategies (SIMD, Packet-level, Instruction-level and data-flow) was carried out. The results showed that the data-flow strategy always performed significantly better than any of the other three strategies. Furthermore, when the performance was measured under heavy loads, the data-flow strategy demonstrated greater performance improvement.

Presented here is an evaluation of the proposed mixed-flow query processing strategy. Specifically, two experiments were performed. First experiment was to compare the performance of the mixed-flow query processing strategy and the data-flow query processing strategy. Second experiment was to test both processing power and buffering requirements of different mixed-flow streaming strategies in order to select the one that is the best compromise between its processing and buffering

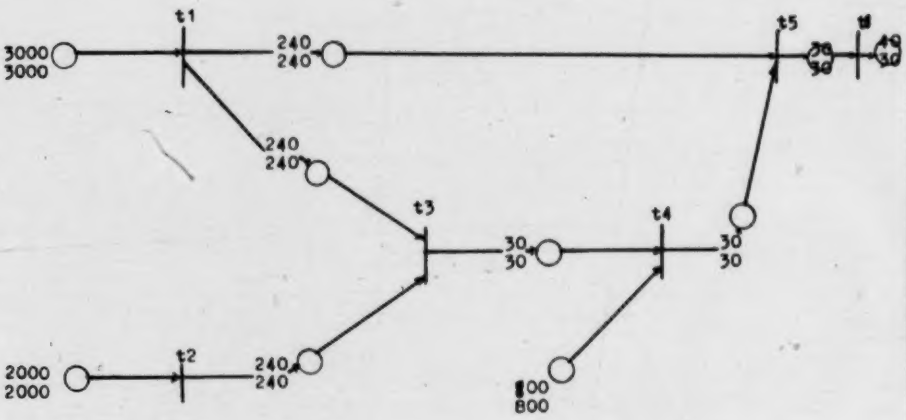
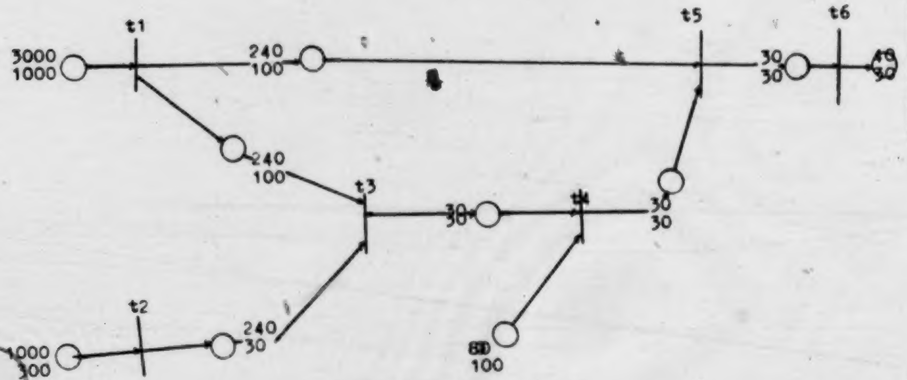

t1
t5
t6
3000
3000
240
240
30
30
40
30
240
240
t3
t4
30
30
30
30
2000
2000
t2
240
240
800
800

Figure 4.5: Data-Flow work-load assignment


t1
t5
t6
3000
1000
240
100
30
30
40
30
240
100
t3
t4
30
30
30
30
t2
1000
300
240
30
80
100

A Mixed-Flow work-load assignment

Figure 4.6

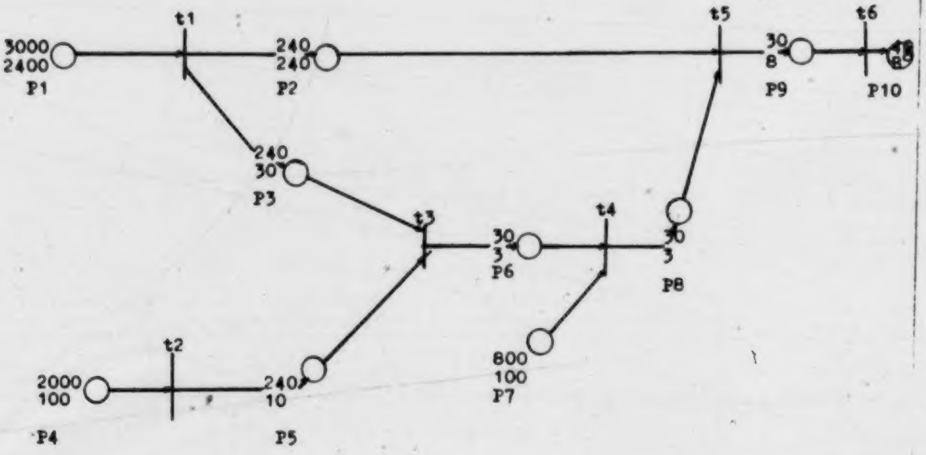
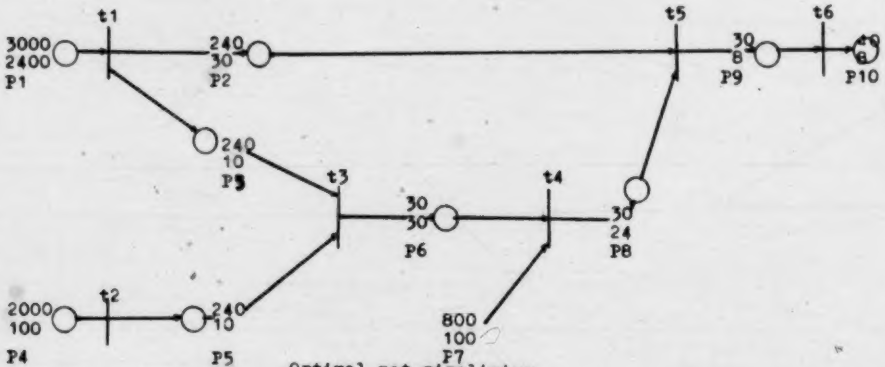

t1
t2
t3
t4
t5
t6
3000
2400
P1
240
240
P2
240
30
P3
2000
100
P4
240
10
P5
30
3
P6
800
100
P7
30
3
P8
30
8
P9
8
P10

Figure 4.7(a)


t1
t2
t3
t4
t5
t6
3000
2400
P1
240
30
P2
240
10
P3
2000
100
P4
240
10
P5
30
30
P6
800
100
P7
30
24
P8
30
8
P9
8
P10

Optimal net pipelining

Figure 4.7(b)

requirements.

For the first experiment, a computer simulation program that simulates the two-query processing strategies was written. This program simulates the execution of both single and multi-query environments. It is monitored to produce statistics which summarize the behaviour of the two processing strategies. For the purpose of the simulation, it is assumed that two types of transitions are available: heavy producers (which have the characteristic that productions > consumptions) such as Joins, and light producers (which have the characteristic that productions < consumptions) such as Selects. These characteristics are represented by what is called "Selectivity factors". Selectivity factors are statistical figures associated with every transition in the net to represent the processing capabilities at these transitions. The single query simulation program enforces two main constraints to allow precise operation. First it associates with each place a capacity (C). Each time a page enters any place in the net, it checks the value of the flow (f) at that place. If this value violates any of the net conservation rules (recall definition 4.8), then this page is delayed to the earliest non-violating time. Secondly, there is an implicit synchronization at net binary transitions; for example each join processing station waits on the availability of its two input source pages before they are hashed as a packet to a specific processor in the Join organization. Beside this implicit synchronization, there is an explicit restriction on processing overlapping (i.e, allowing more than one page to be processed by the same processor at the same time). The multi-query environment is represented by a demand table (which represents several user requests) and a global resource pool. The demand table is traced by a global controller, which serves queries for execution according to first fit strategy (i.e, fitting the amount of available resources). If the amount of current available resources is not enough to initiate a new query, then the global controller either distributes the available pool of resources among running queries (if it is simulating a Mixed-flow strategy) or keeps accumulating them to start a new query. The Global Controller serves users' queries by a Round-robin Dynamic Resource Reallocation service. It runs several copies of the single query simulation model (each running a different query) and provides a communication medium for resource exchange through pipes and shared files. In its realization of resource re-allocation among queries , the simulation uses heaps to rank net bottlenecks (either buffering or processing). Time slices are handled by running several single queries under a controlled environment in which it is possible to interrupt them and change some of the model internal parameters before their execution is resumed. At the beginning of each time slice of each query existing bottlenecks, from the tops of the heaps, are resolved till the exhaustion of the resources at hand. The Global controller continues its Round-robin service by incorporating new queries from the demand table when possible and continually updating the Global Resource pool.

Although the used simulation strategy served our purpose, a discussion of its capabilities as compared to the capabilities of other strategies will be presented at the end of this section. The database used to examine the two query processing strategies consists of five relations. The sizes of these relations were chosen to cover the spectrum of practical relation sizes. Sizes were selected as 800, 500, 400, 250 and 200 pages. For each relational operation the selectivity factor (the fraction of tuples which satisfy the selection condition) was chosen as in Table 4.1.

Two experiments were performed to compare the two query processing strategies. The first experiment served two purposes. The first purpose was to test the effect of buffering assignment to different net places on the total execution time of each query. Whereas testing the two strategies in multi-query environments was the second purpose. Figures 4.9 , 4.10, 4.11 provide indications about the efficiency with which each strategy uses the assigned buffers in order to resolve its bottlenecks (buffer bottlenecks) and improve total execution time.

This experiment was run by fixing the number of assigned processors for each query in Figure 4.8 and by testing the two query processing strategies for different assigned buffering sizes. Figures 4.9, 4.10, 4.11 show that the performance of the mixed-flow query processing strategy continues to improve as the number of buffers increases long after the data-flow query processing strategy improvement has been saturated. If the number of available buffers is increased beyond a certain amount (the amount at which there are no buffering bottlenecks with the existing processing power capabilities) the mixed-flow improvement also starts to saturate.

This result seems to indicate that the mixed-flow query processing strategy indeed succeeded in more efficiently utilizing the re-allocation of available buffers to resolve its buffering bottlenecks. The reason that the data-flow improvement saturated too early is its inadequate global view for the query buffering requirements at different net places.

Furthermore, these results are even more apparent in Figure 4.12, which shows the comparison in a multi-query environment. The figure shows that even for a very large amount of available buffers (about 700 in the figure) the mixed-flow improvement has not yet saturated, while the data-flow strategy has been saturated for a long time (near 250 in the figure).

The reason behind the apparent superiority of mixed-flow strategy over Data-flow in multi-query environments more than in single query environment is its flexibility in inter-query buffering movements, in which unneeded buffers (those that do not contribute to the progress in a specific query) can be moved to another query to achieve a degree of improvement there. Using this technique, an overall average response time improvement is achieved.

QUERYQ1

No. of entry places=1

No. of Selects=3

No. of Joins=2

No. of Projects=1

QUERYQ2

No. of entry places=2

No. of Selects=5

No. of Joins=3

No. of Projects=1

QUERYQ3

No. of entry places=3

No. of Selects=2

No. of Joins=5

No. of Projects=1

Multi-query

No. of Queries=10

No. of class Q1=3

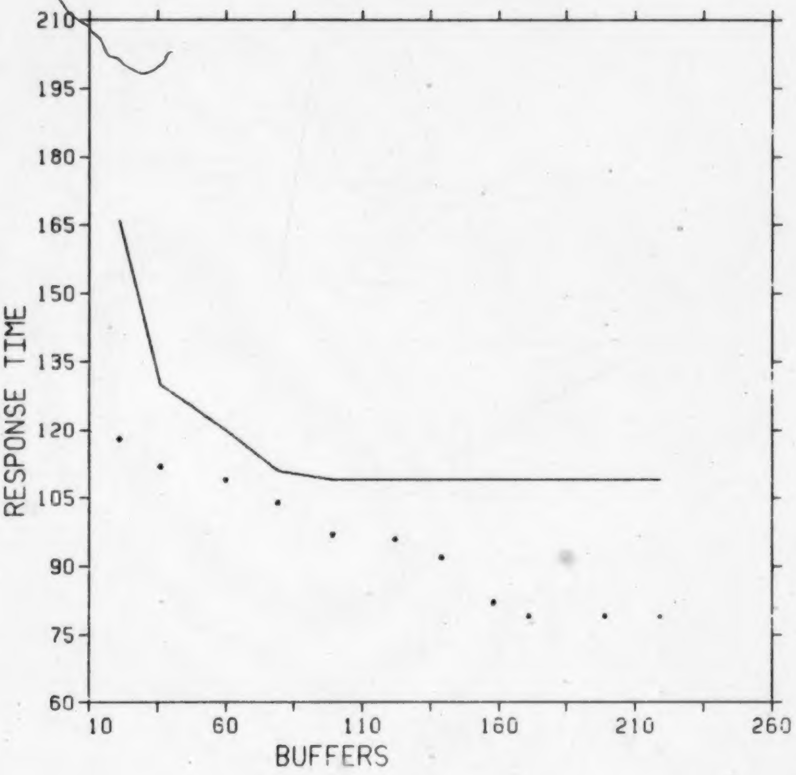
No. of class Q2=3

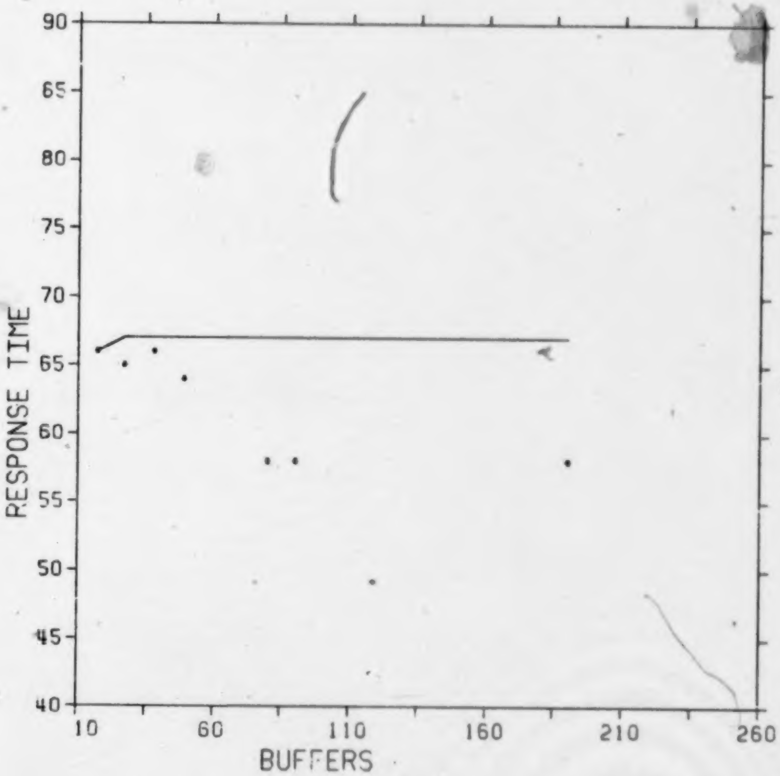
No. of class Q3=4

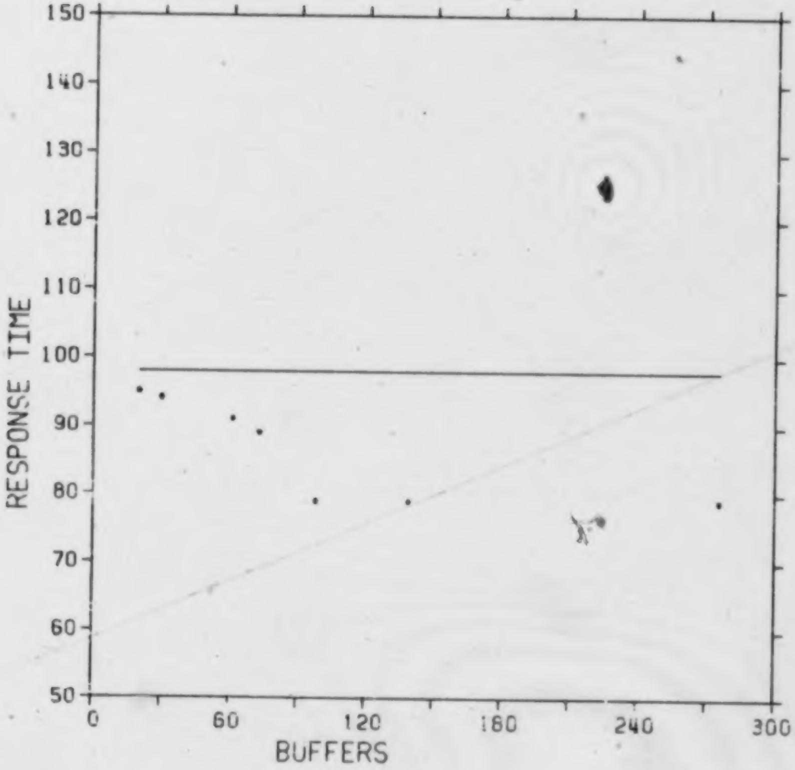
Simulation Sample Queries

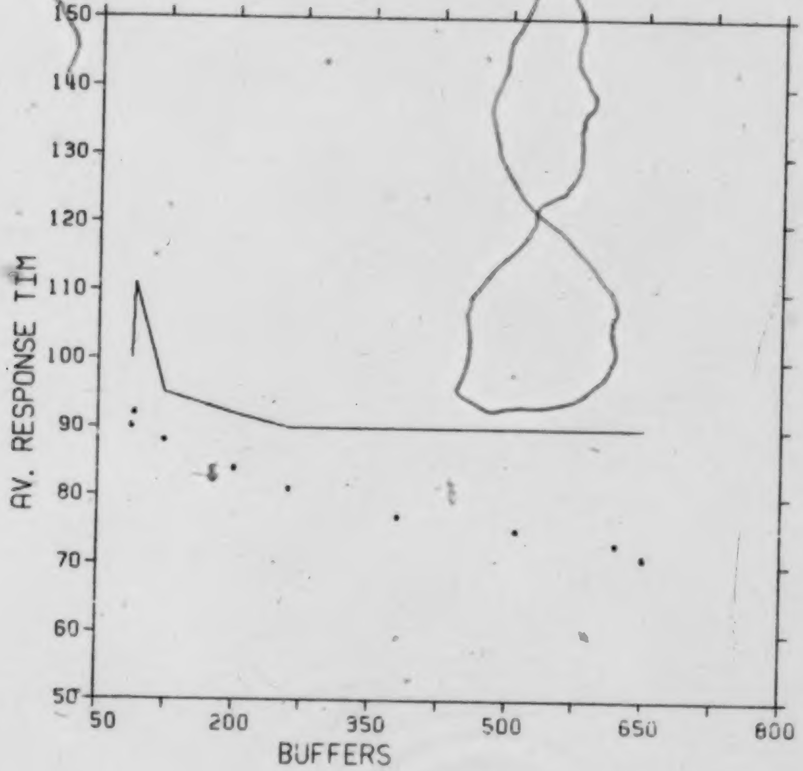
Figure 4.8

University of Alberta


RESPONSE TIME
210
195
180
165
150
135
120
105
90
75
60
10
60
110
160
210
260
BUFFERS


90
65
80
75
70
65
60
55
50
45
40
10
60
110
160
210
260
RESPONSE TIME
BUFFERS


150
140
130
120
110
100
90
80
70
60
50
RESPONSE TIME
0
60
120
180
240
300
BUFFERS


150
140
130
120
110
100
90
80
70
60
50
AV. RESPONSE TIM
50
200
350
500
650
800
BUFFERS

In the second experiment, the three underlying queries were executed using each strategy for a range of variable provided processors. The results of these executions are presented in figures 4.13 ,4.14 ,4.15. For the purpose of the simulation, the x-axis of the figures represents the total net delays (adding more processing power lowers the delays at net transitions). It is clearly shown that, with a specific number of processors added (in order to lower the total delays), both mixed-flow and data-flow strategies show identical response times. At this specific amount, there are no processor bottlenecks in the net, which means that the response time is the same whether or not processors are re-allocated among net transitions. Re-allocation of processors is for resolving processor bottlenecks.

Note that what a processor bottleneck means is the delay that propagates to the net output due to some processing station that could not cope with the work load at its entry place(s). Similarly, buffer bottleneck means the delay that propagates to the net output due to some buffering place that was fully saturated and refused to accommodate a page at a specific instant of the time slice.

Obviously, buffer and processor bottlenecks were expected during the actual operation of the system, since the actual flow (that is based on actual selectivity factors) differs from the planned flow (that is calculated by using the estimated selectivity factors for net transitions) which is assigned by the the proposed procedure. The simulation selects actual and estimated selectivity values and resolves 'old' bottlenecks at the beginning of each time slice.

In fact, the resource re-allocation strategy of identifying net bottlenecks at the beginning of each time slice and resolving them on the tops of the heaps (urgent first strategy is implemented in the simulation), is not an optimal strategy. It always resolves "old" bottlenecks which had occurred in the previous time slices. Any optimal strategy should achieve in addition to the previous goal, a secure future operation (upto a specific amount of time ahead). This means it has to predict future bottlenecks by adaptively studying buffering and processing requirements. To understand the meaning of the "adaptive" study of requirements, consider a single input transition t_i, which has an input place P_i and output place P_{i+1}. Allocating some processing power at transition t_i will increase its consumption ability from P_i and will also increase its production ability to P_{i+1}. As a result, the old buffering requirements of P_i and P_{i+1} are not accurate any more.

This indicates that it is impossible to isolate buffer allocation from processor allocation, since both operations affect each other directly. Any inconsequential change in one could jeopardize the requirements of the other. Most of the previous resource allocation strategies tackle buffer- and processor-allocation independently [Vic81] [Cop77]. Therefore, it is important to develop a global algorithm that studies adaptively these requirements. One such strategy could be:

[1] At the beginning of each time slice, heaps of current bottlenecks(buffers and processing) are built.

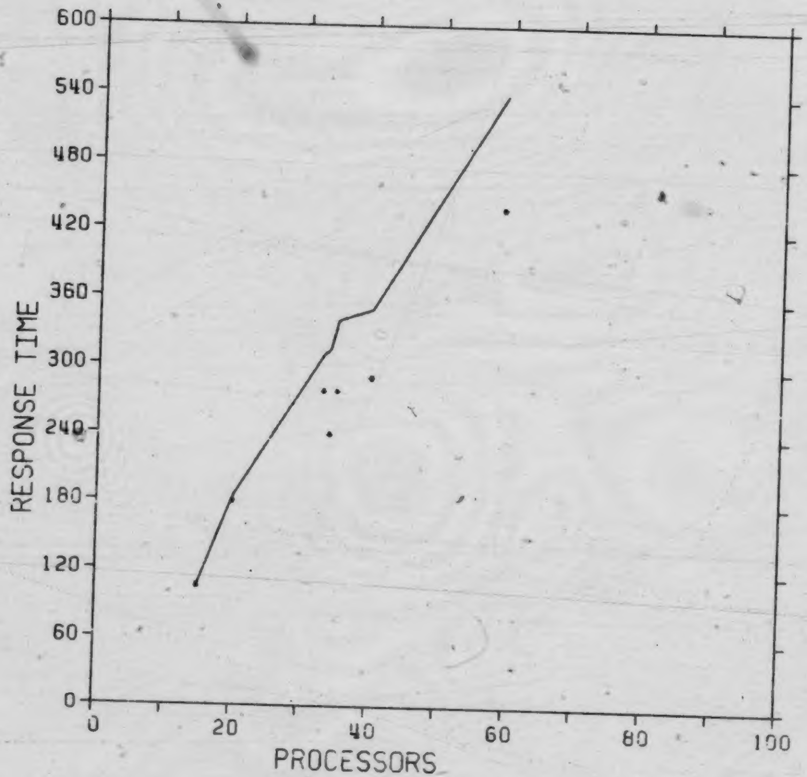
[2] According to the available global resources,a unit of resource resolution is established (e.g, one buffer and one processor).

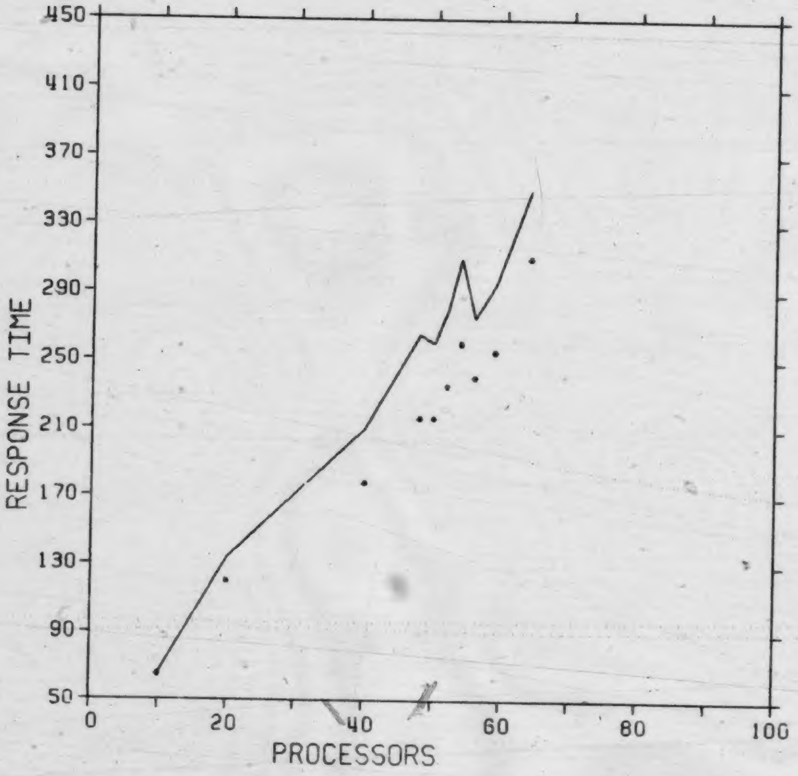
[3] Alternatively one processor bottleneck is resolved followed by one buffer bottleneck using the defined unit of resolution.

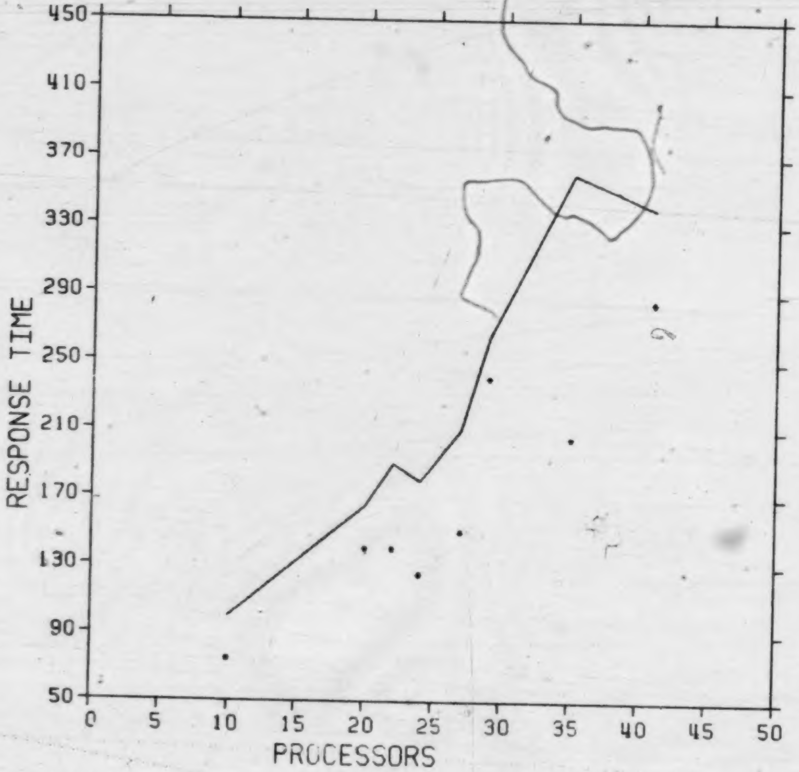
[4] Adjust the heaps by studying the new net requirements.

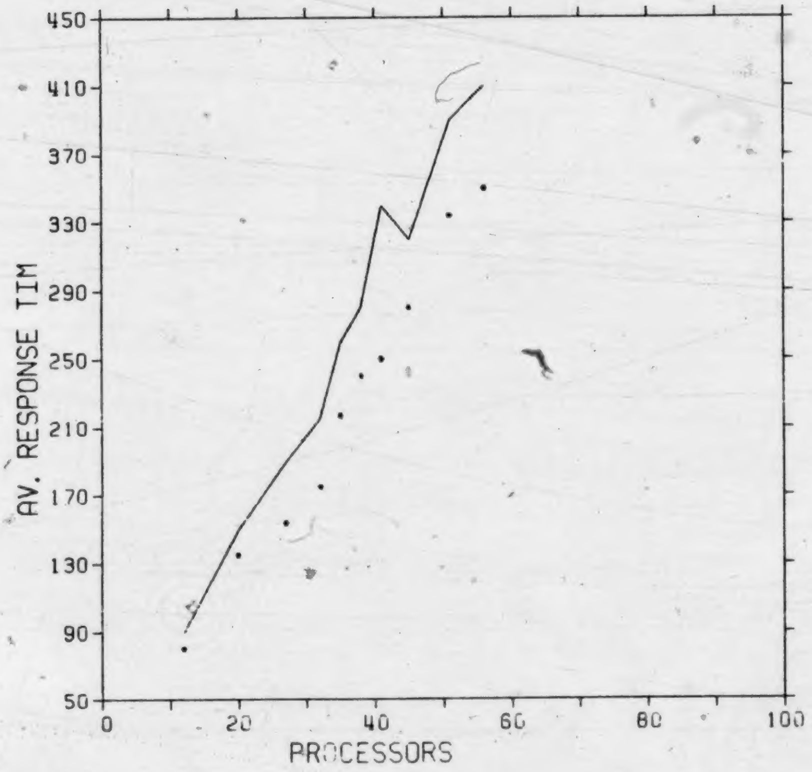
[5] Repeat steps (3) and (4) until either there are no more net bottlenecks or no more resources to be re-allocated.

Note that since the query nets are acyclic (by nature), the previous strategy cannot have any recursive or cyclic effect.


RESPONSE TIME
600
540
480
420
360
300
240
180
120
60
0
0
20
40
60
80
100
PROCESSORS


RESPONSE TIME
PROCESSORS
450
410
370
330
290
250
210
170
130
90
50
0
20
40
60
80
100


RESPONSE TIME
450
410
370
330
290
250
210
170
130
90
50
PROCESSORS
0
5
10
15
20
25
30
35
40
45
50


AV. RESPONSE TIM
450
410
370
330
290
250
210
170
130
90
50
0
20
40
60
80
100
PROCESSORS

Based on the Computation net transition characteristics and input relations information, net processor bottlenecks could be detected at different locations:

[a] In the case of linear pipelines, it is clear that the first processor bottleneck in a linear pipeline path t_i ......t_j is at some t_k in which]l such that the execution time of t_l is equal to the minimum processor execution time in the pipe t_k and $l > k$.

[b] In the case of Forks, the previous identification method needs no change, since the net can be redrawn at the Fork position as a multiple linear pipeline.

[c] Finally in the case of Join, the longest paths from the Computation Space Sink to each Join position could be identified. The bottleneck identification method need only be applied to those longest paths.

The previous experiments indicate the superiority of the mixed-flow query processing strategy in utilizing both processor and buffer re-allocation so as to improve response times. In all the tests (single query and multi-query), the mixed-flow always performed significantly better than the data-flow strategy. More specifically, when one examines the performance of each strategy in a multi-query environment, the mixed-flow demonstrates marginally better performance than the data-flow; this is due to the flexibility of inter-query resource exchange.

As mentioned in the introduction of this section, another experiment was done to test different Mixed-flow streaming patterns. In this simulation, more than having a global control flow and a local data-flow, different types of local data-driven strategies are examined within the time slice itself. The query net in Figure 4.17 shows the 'Forward label streaming' and the 'Backward label streaming' strategies for token number one, starting from the moment it entered the net till it finishes its processing. As shown, the numbers in the places represent the clock times at which those tokens were available at different net places. Obviously, different processing powers and buffering requirements will be needed at different net transitions and places for different types of streaming. Some of these patterns are examined in the simulation, to find the best local streaming pattern. Best means the one that reaches an optimum compromise between its requirements from both processing powers and buffers. The tables shown in appendix B, give figures for the processing powers, and buffering requirements for different streaming sequences at different net transitions and places.

Finally, it is important to mention that the second experiment was also meant to provide a 'Snapshot' information about the details of different mixed-flow patterns. That was meant to be used in a 'graphical' demonstration of these patterns.

```
The Query net:
a               one           c               6
a               one           b               6
c               three         e               6
b               four          d               6
e               seven         f               6
d               seven         f               6
g               two           i               6
g               two           h               6
i               five          k               8
h               six           j               6
k               eight         l               8
j               eight         l               8
f               nine          end             6
l               nine          end             6
a               70
g               25


Token          1
_______________
The following is the forward labelling..
Subquery      Available at    Executable time       Wast time (Awaiting)
a                    0               0                      0
c                    6               6                      0
b                    6               6                      0
e                   12              12                      0
d                   12              12                      0
f                   18              22                      4
g                    0               0                      0
i                    6               6                      0
h                    6               6                      0
k                   14              14                      0
j                   12              14                      2
l                   22              22                      0
end                 28              28                      0
...The following is the backward labelling..
a                    0               4                      4
c                   10              10                      0
b                   10              10                      0
e                   16              16                      0
d                   16              16                      0
f                   22              22                      0
g                    0               0                      0
i                    6               6                      0
h                    6               8                      2
k                   14              14                      0
j                   14              14                      0
l                   22              22                      0
end                 28              28                      0
Token          2
_______________
The following is the forward labelling..
Subquery      Available at    Executable time       Wast time (Awaiting)
a                    0               6                      6
c                   12              12                      0
b                   12              12                      0
e                   18              18                      0
```

University of Alberta

```
d                18          18                   0
f                24          30                   6
g                 0           6                   6
i                12          14                   2
h                12          12                   0
k                22          22                   0
j                18          22                   4
l                30          30                   0
end              36          36                   0
...The following is the backward labelling..
a                 0          12                  12
c                18          18                   0
b                18          18                   0
e                24          24                   0
d                24          24                   0
f                30          30                   0
g                 0           8                   8
i                14          14                   0
h                14          16                   2
k                22          22                   0
j                22          22                   0
l                30          30                   0
end              36          36                   0
```

Figure 4.17:'Forward label streaming' and 'Backward label streaming' for Token one and two.

University of Alberta

CHAPTER 5

MACHINE ARCHITECTURE

5.1. Introduction

Chapter 3 showed how Mixed-Flow architectures can satisfy completely the architecture requirements that serve both the database dynamics and tasks. A number of specialized functional processors to optimally execute database primitive operations and to be used as off-the-shelf components in the target architecture were also proposed. Then Chapter 4 introduced the Mixed-Flow query processing strategy. This strategy utilizes specialized functional processors and overcomes all the problems of the pure Data-Flow query processing strategy. In addition, the Mixed-Flow strategy has the capability to serve in multi-query environments. The computation space model has also been used to investigate several firing (streaming) sequences for the local processing of the Mixed-Flow query processing strategy.

This Chapter will describe the target Mixed-Flow architecture which consolidates all the previous results in a neat and simple design. The design has a unique feature which avoids those problems of processor switching and memory/processor interconnection that usually limit the degree of realizable concurrent processing. The design offers a simple solution to the mapping of queries onto hardware structures.

Section 5.2 looks at the endo-architecture description level of the architecture, giving a description of each component's capabilities in the architecture. Following this a complete execution cycle of the machine is described. Section 5.3 gives a description for program organization; which includes description of machine instructions and data types. Then Section 5.4 presents an example of executing a query on the machine.

5.2. Endo-Architecture Description

It is known that computer architectures can be examined on many levels, from the detailed circuit level to the processor-memory-switch level, developed by Bell and Newell [BeN71]. The most appropriate description level during initial design processes is the endo-architecture level; which is defined by Dasgupta [Das81] as typically including:

> "...the functional capabilities of a machines' physical components, their interconnections, the nature of the information flow between components, and the means whereby this flow is controlled".

The following subsections consider the architecture at its endo-architecture descriptive level. The architecture organization is described first, followed by a description of a complete machine execution cycle.

It is a well known fact that in order to obtain high speed from any parallel computer system, it is necessary to exploit parallelism in processing, storage and information transfer. The critical "bottleneck" found in most MIMD machines usually appears in the form of crossbar switches (e.g, DIRECT), common highways, or the common stores through which all processors in the system may wish to communicate. The reason for this can be traced to the need for a processor to rapidly acquire data from any other part of the system. In addition, there is the necessity of controlling access to data which has yet to be formed; this can also introduce significant communication overheads.

The scheme used by data-driven architectures to alleviate the previous troubles is to not require the processor to perform a section of a computation until all the data are presented to it as an executable packet. Rapid data access to other parts of the system and access control are then unnecessary.

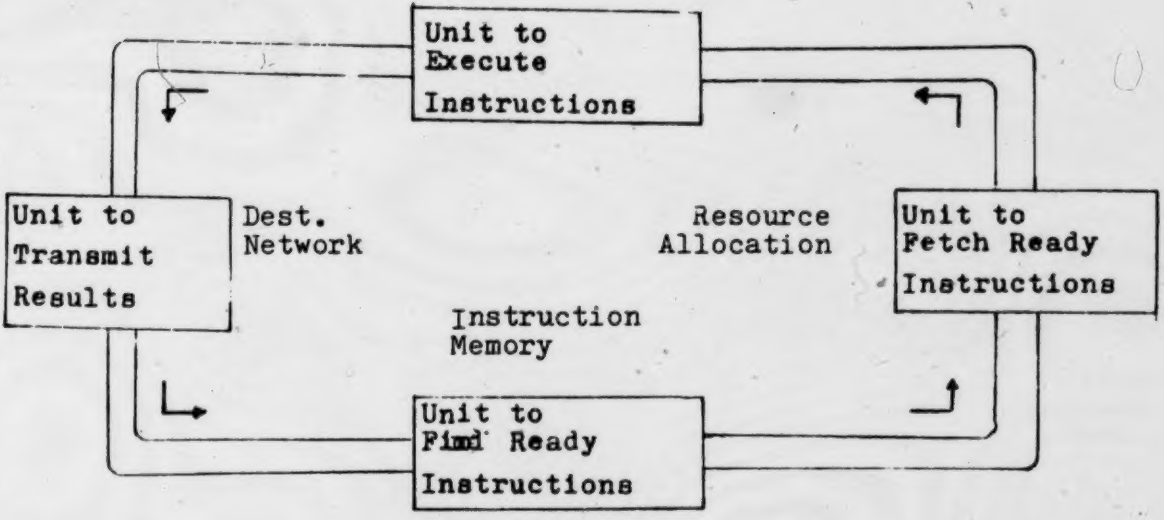

Unit to Execute Instructions
Unit to Transmit Results
Dest. Network
Resource Allocation
Unit to Fetch Ready Instructions
Instruction Memory
Unit to Find Ready Instructions

Figure 5.1
Data-Flow Architecture
Scheme

Basically any data-driven architecture consists of the four primary components found in Figure 5.1 [Bae80]. A data-flow scheme to be executed is stored in the memory of the machine (the instruction memory) in a form of instructions. Each instruction corresponds to an operator of the data-driven program. Each instruction contains fields to specify the operation to be performed and the address(es) of the instruction(s) to which the result of the operation is to be directed. Other fields hold the operands for use in execution of the instruction. When an instruction contains all the necessary operands, it signals the 'Resource Allocation' Unit (see Figure 5.1) that it is ready to be transmitted as an operation packet to a processing station which can perform the desired function. The Resource Allocation Unit is responsible for directing operation packets to appropriate processing stations. The results of a processing station leaves it as one or more data packets. Data Packets are composed both of fields to hold the results, and of other fields for the address(es) of the instructions in the Instruction Memory, to which these results are to be delivered. The Distribution Unit receives data packets from processing stations and decodes the address(es) of each to direct the result items to the correct instruction(s) in the Instruction Memory. The instruction(s) receiving those values may then be enabled if all operands are present in them and the cycle is repeated again.

The previous scheme is basically the execution cycle of any data-driven machine. Different data-driven machines may vary in how they implement this strategy. For instance, many instructions may be enabled (signaled for execution) simultaneously, at a time where there are few processing stations to cope with that demand, in such cases, different scheduling strategies are employed by different data-driven machines to define fair 'fetching' polices among instructions. Other strategies are also implemented in the Distribution Network to match result propagation demand.

The proposed database machine, shown in Figure 5.2, follows the same lines of data-driven architectures: it has an Instruction Memory, a Resource Allocation Unit, a Processing Stations Unit and a Distribution Network. These Units are performing the same functions, as described in the general data-driven architectures, with the following differences:

[1] Instead of storing operand values in fields within the instructions, it is decided to utilize single and double queues, for single and double instructions respectively to store operand addresses. The reason for this particular change is that most existing data-driven machines were developed mainly for numeric applications in which operands are usually scaler values or at most vectors, so fixed length fields in the instructions are capable of storing these operands. On the other hand, in case of relational databases these operands are going to be pages of variable length tuples, and it is inefficient to store relational pages within the instructions

themselves. Furthermore, using the idea of the address queues will allow us to process many invocations ahead, and to store the address of each result in its order in the queue (implementing dynamic concurrency), whereas storing the operands in the instructions, means modifying the program code, thus preventing the support for program code re-entrance, and restricting each instruction to store no more than the current operand [DLM77]. The sizes of these queues could be precisely calculated knowing the 'streaming' type used in processing the query and the global work load assigned to the query instructions (look at the simulation results). It is apparent that there is an analogy between the propagation of pages through the query net elementary transactions in the mixed-flow query processing strategy (described in Chapter 4), and the propagation of addresses through the instructions in the machine.

[2] Consequently, the execution cycle had to be changed by storing relation pages in a separate cache memory, called the 'Buffering subsets' Unit, and by providing parallel storing and retrieving capabilities using a distributed intelligent 'Page Management Unit'. As will be shown this 'Page Management Unit' has the responsibility of delivering relation pages to processing stations according to the well-defined matrices mentioned in Chapter 3 (see figure 3.7) when executing the broadcasting Join or Project operations. This shows that the unit is performing page swapping in an intelligent manner [MKY81].

[3] The distribution Network is not receiving result pages from the processing stations. These result pages are first stored in appropriate buffers, and their locations, instead of their original values, are returned back with destinated instruction address(es) to the Distribution Network. The Distribution Network routes these 'modified' data packets from the Buffering Unit to the instructions specified by the destination addresses: This is done by decoding the packets

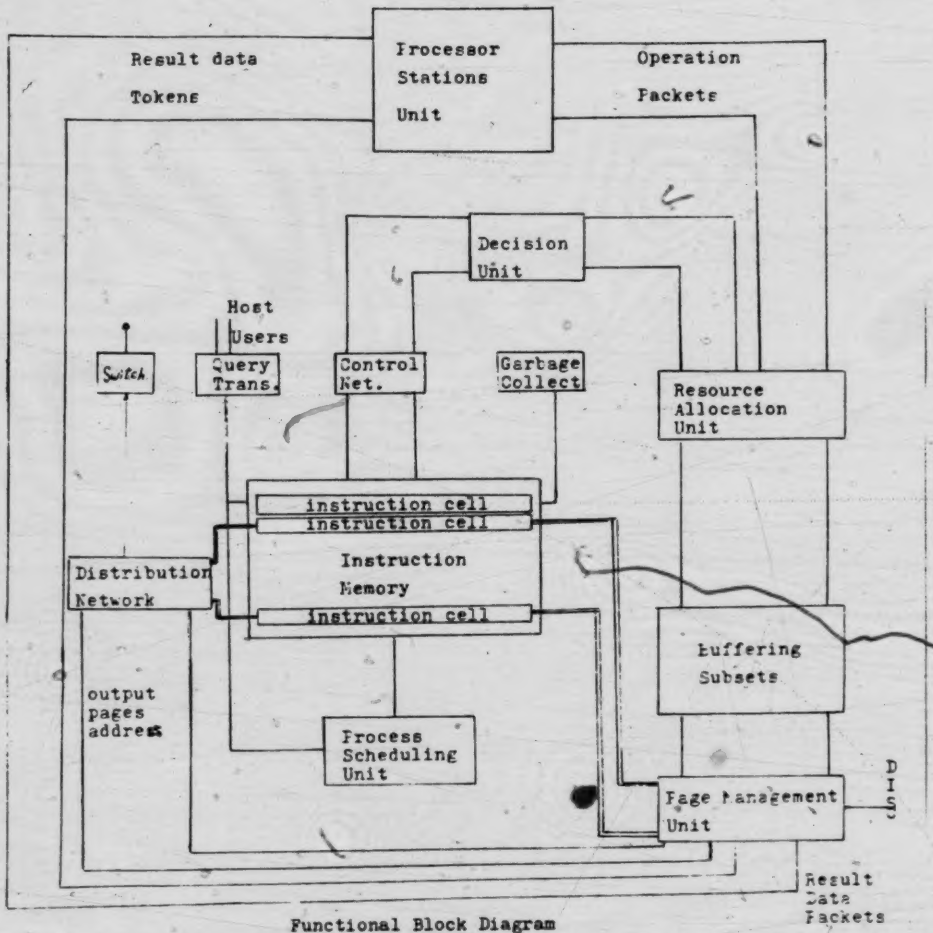
according to their destination address field(s). (The structure of these data packets will be described in the next section.).

[4] The instructions cannot signal the Resource Allocation Unit that they are ready to be transmitted as operation packets to processing stations since they have only operand addresses (not the real values). Therefore, another modification to the execution cycle is to forward instructions to the Page Management Unit to fetch their operands before requesting execution from the Resource Allocation Unit. This action completes the extension of the execution cycle such that any I/O to and from the Instruction Memory should pass through the Buffering Unit to exchange pages with address, and vice versa. Once the instruction fetches its operands from the buffers, it constitutes what is called an 'Operation Packet'. An Operation Packet consists of the instruction with its real operands. A more detailed description of the Operation Packets and other machines' Packets will be provided in the next section.

In addition to the previous modification to the data-flow main execution cycle, it is found necessary to establish other units for different purposes:

[1] The Process Scheduling Unit:

This unit was introduced to regulate the data-driven nature of the machine. It has the main rule of the global work load assignment, mentioned in the mixed-flow query processing strategy in Chapter 4. It distributes time slices among current Computation net active queries and assigns global work loads to their elementary transactions. The algorithm of maximum net flow is implemented here as a mechanism with which the Process Scheduling Unit can calculate the amount of sufficient work loads and assign them to transactions. As will be shown in the next chapter, the Process Scheduling Unit can also be used to manage the swapping of instructions to and from the Instruction Memory in case of having a hierarchical memory for storing


Processor
Stations
Unit
Result data
Tokens
Operation
Packets
Decision
Unit
Host
Users
Switch
Query
Trans.
Control
Net.
Garbage
Collect
Resource
Allocation
Unit
instruction cell
instruction cell
Instruction
Memory
instruction cell
Distribution
Network
Buffering
Subsets
output
pages
address
Process
Scheduling
Unit
Page Management
Unit
DISC
Result
Data
Packets

Functional Block Diagram
of the Proposed Architecture
Figure 5.2

instructions. Such memory hierarchy will be suggested as one of the possible extensions to the proposed architecture in Chapter 6.

The Process Scheduling Unit has another important function, which is to update the current Computation Space Net. Updating the Computation Space Net means moving processing power from place to another, in order to start executing a new query, or to further push more pages towards active queries sink point (resolving bottlenecks). The net updates, described in Chapter 4, form the tools by which the Process Scheduling Unit moves processors and/or buffers from a specific query net (because they have already finished their share in processing the underlying query) to participate in a progress undertaking at another part of the current Computation Space net. This service is really needed as the mixed flow query processing strategy is putting such a scheme into effect. The idea is very simple: suppose two queries, A and

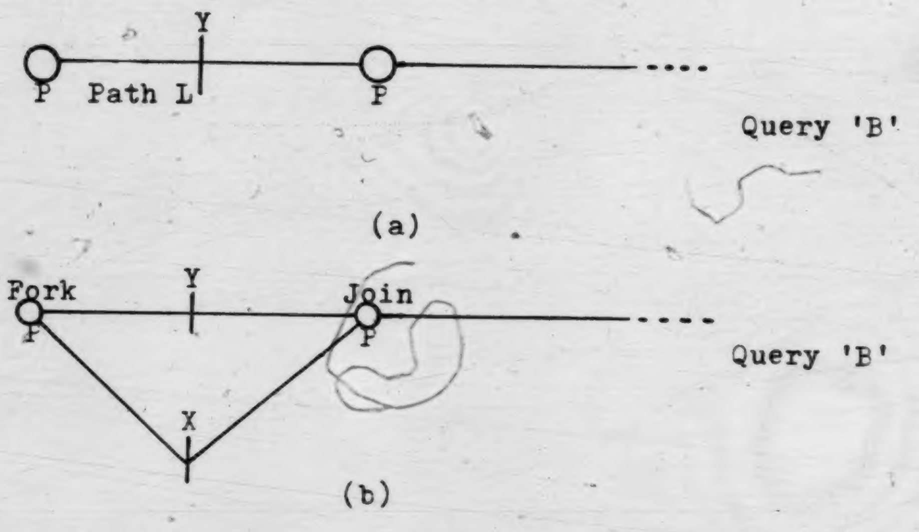


Figure 5.3: Net updating

B, were running concurrently with specific time slices assigned to each. According to the mixed-flow query processing strategy, assume a select processor station, X, has finished its share in executing query A. Now it can be utilized to contribute to any progress undertaking in the current Computation Net. This progress may be in query A itself or in another query such as B. The critical paths along any of the queries are simply identified by the Process Scheduling Unit (according to actual flow and net bottlenecks). Suppose query B has path L ,shown in Figure 5.2, as a critical path (a path that has one or more buffer or processor bottleneck) with a processor bottleneck at processing station y. Now, since the execution is carrying on that path the Process Scheduling Unit updates it as in Figure 5.3(b). It converts the place P_i into a Fork place and adds the X processing station, and converts the other place ,P_j, into a Join place. By doing this, the long waiting line of pages at P_i can be executed by both the old working station and the added one, probably resulting in improved response time. The Process Scheduling Unit keeps up-to-date information about the active Computation net behaviour. Each Instruction cell in the current active Computation net reports the actual flow status to the Process Scheduling Unit. For example, the finishing of the select processing station X to its share in executing query A would have been reported by the Instruction cell that contains this select instruction. These net behaviour informations are used to adjust the active Computation Space net bottleneck heaps such that the Process Scheduling Unit would know where to forward X. Resource re-allocation, then,takes place in two situations: (1) Each time the Process Scheduling Unit receives a 'finishing share' report from some instruction cell in the current active Computation Space net; and (2) The action of initiating a new query net after receiving a 'finishing share' report establishes the situation of having several overlapped time slices in the system. This suggests performing a round of resource re-allocation before starting any new time slice. The benefit here is that existing bottlenecks are resolved before initiating any new time slice.

The Simulation of Chapter 4 implements only the second type of resource reallocation.

[2] The Control Network:

This network was introduced to facilitate imposing concurrency control mechanisms on the current Computation Space net(i.e, the instructions which are stored in the Instruction Memory). The Control Network receives boolean data tokens (called control tokens), and activates/deactivates different parts of the Computation Space Net, according to the concurrency control instructions delivered to the unit through the Decision Unit. This will be apparent when describing the special gate fields in the machine primitive instructions which are associated with instruction operands, and are used to permit or prevent the instructions to be delivered to the Resource Allocation Unit as ready packets for execution.

[3] The Decision Unit:

This unit provides a straightforward solution for the incorporation of decision capabilities into the machine (to serve the generalized query net model as well as the basic query net). The unit has the responsibility to interpret instructions that represent decisions in the Computation Space net and yield control packets (analogous to data packets). These control packets have a boolean field to hold the result of a choice, and other fields to specify the destination primitive instruction address(es) that they are going to affect. The control packets are forward to the Control Network to allow for gating and non-gating of machine primitive instruction operands.

[4] The Query Translator:

This unit has the responsibility of driving query nets from queries by applying the algorithm given in Chapter 3. Several users can interact with the machine through this unit, which compiles user's queries and integrates them into current environment Computation Net. In fact, this unit has the same function as the INGRES parser, which has been used in DIRECT [DBF80] to transform users' queries into binary trees.

[5] The Garbage Collector:

This unit has the responsibility of tracing the instruction memory and swapping out all the instructions that have finished their share of workload. The unit is similar to the Garbage Collector used in CASSM, which runs all the time to delete the flagged (flagged for deletion) tuples. The unit is to be utilized in the extended machine which will have another back memory for storing machine Instructions.

[6] The Processing Stations Unit:

The specialized functional processors proposed in Chapter 3 are incorporated in this unit with two facilities to allow for processors grouping:

[a] Since in some primitive operations (such as join and project) it is needed to group some processors to form a station (organization of processors), there should be some sort of interconnection among each primitive processor type. This is necessary in order to create a means to build a Join Processing Station (a ring interconnection type) or a Project Processing Station (a two way unidirectional connection). In this machine, each primitive processor type is connected using a crossbar switch. This allows any pattern of data exchange among processors of the same type with almost no communication overhead.

[b] A "Decomposer" and a "Composer" units are introduced to precede and follow each group of organized processors. These units have the following functions: the Decomposer is used to manage the organization; in addition , it has its local memory (which may be in one of the buffer subsets) to accumulate the incoming pages, in case there is a need for multiple phase operations (this will be apparent in the example of Section 5.4). Furthermore, the Decomposer, with the help of the Page Management Unit, keeps a table for each processor, and delivers pages to it according to the well-defined matrices described in Chapter 3. The function of the

Composer is to conform with next processing station for the correct delivery of its station output. It also has other functions, such as page compression after delete or duplicate elimination operations, which are likely to create page 'fragmentations' as noted by DeWitt [BoD81].

As the Processing Stations cannot perform arithmetic operations such as sum, average, etc., additional arithmetic processors are available as another processing type in the processing unit.

[7] The Switch

The Distribution Network directs the results to the host or the users through this unit.

The Resource Allocation Unit is a switching network which receives executable operation packets from the Instruction Memory and sends them to the appropriate processing station in the Processor Stations Unit. The Distribution Network is another switching network which receives data packets and sends them to the Instruction Memory. The Instruction Memory receives data and control tokens and stores the appropriate information into instruction cells. All communications among these units are asynchronous.

This structure introduces parallelism by allowing each cell in the Instruction Memory concurrent access to the Processing Stations Unit through the Resource Allocation Unit. Parallelism is also introduced by the specialized processing stations in the Processing Stations Unit which allow concurrent execution of different primitive operations.

5.3. Machine Instructions and Data Types

The machine is designed to utilize the generalized query net model as its base language. An environment (multi-users' jobs) in the current Computation Space net consists of two types of instructions: instructions that correspond to primitive database operations, and instructions that correspond to conditions and choices. Consequently, two types of tokens are available in the machine: data tokens, which represent relation pages that flow through database primitive instructions, and control tokens, which represent the results of logical decision instructions. Control tokens direct the flow of the data tokens by means of gates which exist as fields within the primitive instructions. These control tokens are the output of the condition and choice instructions which are processed by the Decision Unit. Data tokens are produced by the Processing Station Unit which deals with the first type of instructions. Two formats for primitive operation instructions are shown in Figure 5.4, one for unary primitive operations (those to be executed on single buffer processors), and the other for binary primitive operations (those to be executed on double buffer processors). The first field within any token or packet in the machine is used to identify its type. The following codes are used:

I → Instruction
OP → Operation Packet
CP → Control Packet
D → Data token
C → Control token

The remaining fields in the primitive operation instructions are:

[1] Operation Code: This specifies the primitive operation executed by this instruction.

[2] Destination Address: This field identifies the target instructions for the packet(s) generated by the instruction execution.

[3] Operand queues: For each operand there is an operand queue, (q_i), which is a fixed size queue used to store the operand address(es) of the instruction. The sizes of these queues indicate the maximum amount of subsequent work ahead that can be done. Each operand queue is controlled by a gating field ,(g_i), which shows the

I	Op.Code	Dest.addresses	g1	q1	g2	q2	Iter.	Source	Duties

I	Op.Code	Dest.addresses	g1	q1	Iter.	Source	Duties

Figure 5.4: Machine primitive instructions

permission type that is associated with the top operand of that queue. The permissions are assigned by the Control Network, as mentioned earlier. The types of permission are: (a) the associated top operand is nonconditional,and (b) the associated top operand is not allowed to be used at the moment by the instruction.

[4] Iteration Number: This field represents the index of the operand being processed by the instruction (there may be two indices in case of binary instructions). These indices are reset at the beginning of each time slice.

[5] Source Name: This contains the address of one or more source instructions that precede this instruction in the net.

[6] Duties to be Executed: This field is filled by the Process Scheduling Unit, according to the net maximum flow calculations. A comparison between the value of the Iteration Number Field and the Duties to be Executed Field is done by both the Process Scheduling Unit and the Garbage Collector Unit. The later, as

already explained in the previous section, looks for instructions that have already finished their share of work load in order to swap them out.

The two data token types, which are handled by the machine primitive instructions, are shown in Figure 5.5. The data tokens produced by primitive stations have four fields:

[1] Identifier: 'D' indicates that the token is a data token.

[2] Output Page: This field contains a relation page.

[3] Destination Address: This field specifies the target instructions for receiving this data token as an operand.

[4] Source Name: This specifies the source instruction that generated this token.

The other data token type in Figure 5.5 has three fields, the first and third fields are the same as in the previous data token type. The second field is the Page Address

---

D	Output Page	Dest. Addres.	Source Name

D	Page Addres.	Dest. Addres.

Data token types
Figure 5.5

in the Buffer: this field contains the address of the result page. It is filled by the Page Management Unit, as described in the previous section. This data token type is formed out of the first data token type by the Page Management Unit and is directed to the Distribution Network Unit.

The other type of instructions is the Choice instructions type, which has the following fields:

[1] Identification: This field is used to identify the type.

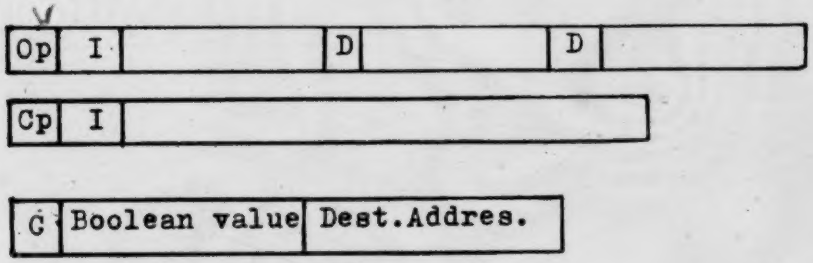
[2] Predicate: This contains the predicate on which the choice is based.

[3] Destination Address: This field specifies the target instructions to be controlled by the packet(s) generated from this instruction execution.

[4] For each operand, there is a value field (v_i) associated with another gate controlling field (g_i). The function of the gate field is the same as described before.

There is only one token type produced by the previous choice instructions: a control token. Figure 5.6 shows the structure of a control token. The leftmost field is used ,as usual, to identify the token type; the next field is used to store the Boolean results (true/false), and the right-most field is used to specify the target instructions which are going to receive this data token as a controller over its operand gates.

Finally, Figure 5.6 shows the two types of Packets the machine has: Operation Packets and Control Packets. Operation Packets are sent to the Processing Stations Unit , while the Control Packets are sent to the Decision Unit. Both Packets are sent through the Resource Allocation Unit. The Operation Packet consists of the instruction to be executed associated with its real operands; it has one extra field used as token identification.



Operation and Control Packets

Figure 5.6

The Control Packet simply contains a choice instruction (since it has its operands within the instruction).

5.4. Query Execution Example

Figure 5.7 shows how to translate a query net into executable machine instructions to be loaded into the machine's Instruction Memory. Also, three different data tokens are shown ready to flow through the net. Figure 5.8 shows a logical picture of the interconnections within the Processor Stations Unit; it shows the rings of Join Processing Stations and the two-way unidirectional bus of a Project Station. Processors are assigned to every station by the Resource Allocation Unit, which determines the number of processors in each station.

The places in that figure represent buffering subsets. This example, for the sake of simplicity, does not show any choice or condition transitions. The figure is largely self-explanatory; some points are presented below for clarification:

[1] For a Select instruction: One select processor or more is assigned to this instruction; this is decided by the Resource Allocation Unit.

[2] For a Join instruction: The parallel nested loop Join procedure, described in Chapter 3, relies heavily on the broadcasting facility; it works as follows:

[a] The Resource Allocation Unit determines the optimal number of Join processors to participate in the execution.

[b] The Decomposer hashes each received operation packet, according to the address of the inner relation page, to a certain processor within the organization. Then the concerned Join processor issues a READ command to enter both operands in

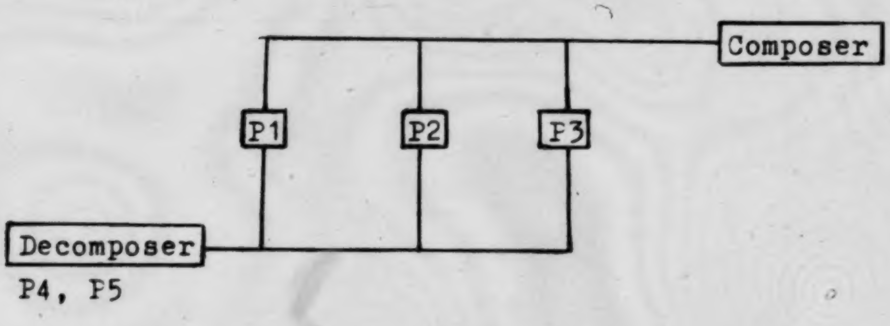
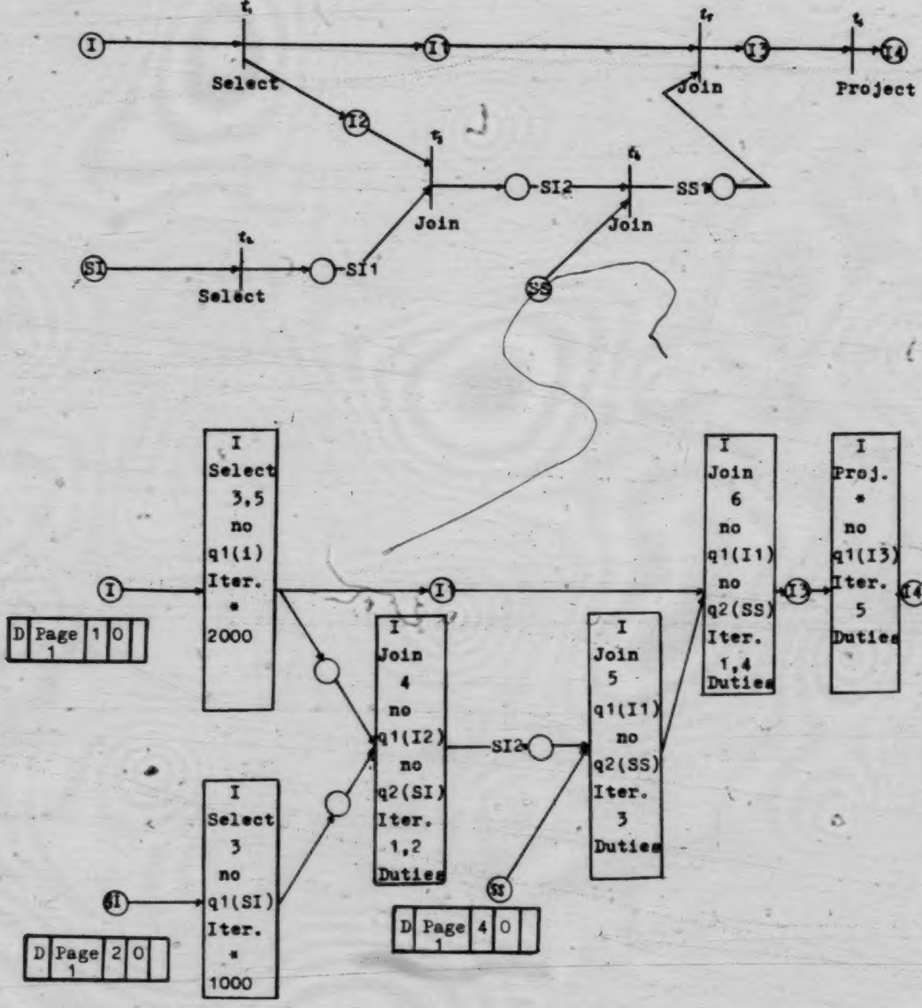


Figure 5.9: Project processing example

its internal buffers. At this time, every other station's Join processor broadcast its outer relation page to other stations' processors following the well-defined matrix of the Join. (Notice that the Join is completed when all pages of the outer relation are seen by every page of the inner relation).

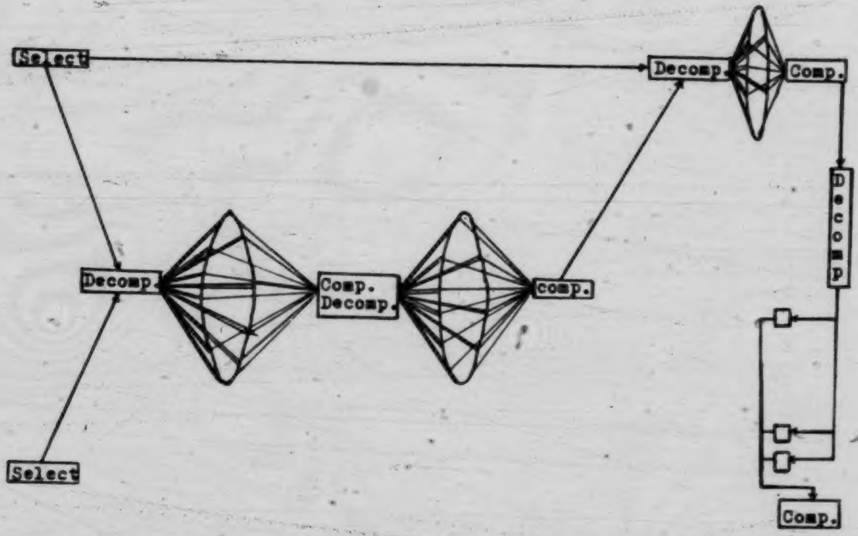
[c] The output tuples from any stations' processor are moved directly to the Composer, which collects them and produces result data tokens for the Distribution Network. (Note the output pages are delivered, as soon as they are ready, out of the station).

[3] For Project Processing Station: The Decomposer reduces the source relation to a "vertical" subrelation by discarding all domains that are not needed. Since discarding attributes may introduce duplicate tuples, the duplicates must be removed in order to produce a proper relation. Each P_i initially deletes the inter-page duplicates by copying the contents of its ALAP into its input buffer, then matches a tuple at a time and deletes duplicates. Then each processor, in turn, broadcasts its page through the upper uni-directional bus and exits (to output its contents to the Composer through the other lower uni-directional bus). If processor P_j receives page i, then j>i. P_j compares the two pages and eliminates any duplicates found from its page(j). Note that P_j will not see page i if i>j (because of the undirectional bus). Consequently it is guaranteed that only one copy of each tuple will remain in the relation (the copy will reside in the lowest numbered page of all the pages that had a copy of it). To see more closely how the projection processing station works in the context of a complete query net, assume that five pages produced by the selection S1 (in Figure 5.9), and only three projection processors are constituting the project processing station Pr1. Therefore, two phases will be required to execute the project at this elementary transaction. The sequence of actions is as follows:


I
t1
Select
I1
I2
t3
Join
SI
t2
Select
SI1
SI2
t4
Join
SS
SS1
t5
Join
I3
t6
Project
I4
I
Select
3,5
no
q1(I)
Iter.
*
2000
D Page 1 1 0
I
I
Join
4
no
q1(I2)
no
q2(SI)
Iter.
1,2
Duties
SI
I
Select
3
no
q1(SI)
Iter.
*
1000
D Page 1 2 0
SI2
SS
D Page 1 4 0
I
Join
5
q1(I1)
no
q2(SS)
Iter.
3
Duties
I
Join
6
no
q1(I1)
no
q2(SS)
Iter.
1,4
Duties
I3
I
Proj.
*
no
q1(I3)
Iter.
5
Duties
I4

Query net example

Figure 5.7


Select
Decomp.
Comp.
Decomp
Decomp.
Comp.
Decomp.
comp.
Select
Comp.

Query net of processing Stations

Figure 5.8

[a] The Resource Allocation Unit assigns the three projection processors for the Project instruction, and the Decomposer assigns identification numbers to those processors for the duration of the instruction execution (to be used in the hashing).

[b] A message exchange for distributing the outer relation pages is done by the Decomposer which receives those pages and hashes each page to a specific processor in the station. Every page i is read by the Decomposer ,and a processor P_i. Since, in this example, the number of pages is greater than the number of participating processors by 2, only the Decomposer will receive the additional pages and will store them in its temporary storage. These pages will be distributed among processors in subsequent phases of the execution of this project operation.

[c] Following this, the broadcasting step starts, with the Decomposer broadcasting pages 4 and 3. Each P_i will read the pages, one at a time, and search for duplicates on its own page and eliminate them. Following the projection algorithm, P_2 broadcasts its page followed by P_1 and P_0. Now P2's page is purged of any duplicates and can be used in the subsequent operation (the output to the user in our example). P_1 executes the same procedure after eliminating duplicates between its page and page 2. Finally, P_0 is the last processor in the group to broadcast its page. P_0 then produces its page as output, after eliminating duplicates between it and page 1. Both P_1 and P_0 will stay in the organization for the execution of the second phase of the projection. The steps in this phase are the same as in the previous phase and hence need no explanation.

[d] The Composer stores all the output pages produced by any P_i in the station. After finishing all processing, it checks with the next processing station as to whether it has received the correct number of pages, and makes back up copies of

the missing pages if necessary.

It should be noted that the execution of this processing station is proceeding independently and concurrently with that of other net processing stations.

CHAPTER 6

Conclusion

6.1. Contribution and Consequences of the Research

The work reported in this thesis contributes both to the understanding of query processing strategies in multi-processing environments and to the development of database machines. The design of the proposed architecture showed that some form of data-flow scheduling can indeed improve performance. However, the pure data-flow scheduling in a pure data-flow architecture proved to be inapplicable to database environments.

One of the prime objectives of this research was to design an architecture that supports equally efficient execution of all types of database queries. The development of the architecture in the thesis was carried out in a stepwise, systematic fashion to demonstrate that the majority of the architectural features were introduced with the objective of providing such equally efficient execution support. It should be obvious from the thesis that no unnecessarily complex mechanisms were introduced just for the sake of innovation.

The underlying rationale behind introducing the mixed-flow notion is that it is observed that in the pure data-flow query processing strategy, queues of partially completed computations are relied upon to keep the machine busy (such as the uncontributing 900 pages in the example of section 4.4.1).

If only keeping machine execution units busy during query execution will keep its execution time at the minimum, time would be better spent realizing this option. Unfortunately, there are reasons to believe that this would not lead to minimum execution time (especially in multi-query environments): (1)The need for immediately executing an instruction whose result is critical for the execution of other instructions will always exist. The selection of this instruction among other ready instructions is probabilistic, since non-deterministic selection schemes are always employed. (2) The unregulated order of instruction execution will create queues of partially completed computations which absorb some of the resources required for parallelism.

Regulating the flow beneficially to overcome the previous problems can be achieved by introducing mechanisms in the pure data-flow query processing strategy. Planning the processing flow in advance and adaptively re-allocating resources around the processor network are two mechanisms introduced in this thesis.

The transformation to a maximum-flow/minimum cut allowed us to plan contributing flows at different net processors and realize the optimal degree of network pipelining. The adaptive re-allocation of resources provided a secure future net operation (as close as possible to the planned flow) in addition to resolving 'old' network bottlenecks. (Several other regulations to the flow can be found in a previous work [BoD83].)

As well as preserving parallelism, these regulators are all feasible from the architecture point of view. For example, according to the classification of mixed-Flow architectures [Haz82], it is possible to categorize the proposed architecture as:

[a] Semi-automatically Synchronized:

Within its local processing it performs in a fully automatic synchronized manner, but there are instants in which operands are available within instructions but are not allowed to fire. This may happen due to one of two situations: either the assigned amount of work load has been exhausted by this instruction ,or a gate of some operand is imposing its concurrency control (permission) on that instruction.

[b] Run-time ordering for execution:

The total execution order is determined by the "Resource Allocation Unit". Other external factors such as the concurrency control mechanisms are imposed by the control Network to define a secure total ordering.

[c] Allocation of Resources:

Both buffer and processor allocation are being changed during run time.

The proposed architecture exhibits several desirable features. The following summarizes which of the desired features (those presented in Chapter 3) have been achieved in the new architecture.

[a] Clearly, the machine has employed the primitive processing stations proposed in Chapter 3.

[b] The broadcasting capabilities of the crossbar switch in DIRECT are still retained although a different local and more easily expansible, implementation for the interconnection device has been realized.

[c] The machine, with its logical instruction/processor and page/buffer mappings, provides a great support for MIMD activity environments, without making any part likely to become a bottleneck.

[d] The mixed-flow query processing strategy described in Chapter 4, is being employed by the machine, and hence an exact mapping of the environment's current demand onto the hardware structure of the machine has been realized. This avoids the wasted "fragments" of resources which are likely to be unused in such multi-query environments.

[e] The act of updating the computation space net, by moving processing power between different net parts, clearly satisfies the self-adjusting requirement sought from the design. This achieves the goal of improving the average response time in

a dynamically changing environment.

[f] The back-end controller of any MIMD machine such as DIRECT is its only controlling processor; thus when a number of queries are active, the controller is a bottleneck [BoD81]. In the proposed architecture this cannot happen because the control of each instruction is overseen by a different processing station. Also the control within compound processing stations (such as Join and Project) is overseen by different Decomposers and Composers.

[g] The Computation Space net model has been implemented as Instructions in the instruction memory and parts of it are swapped in and out; this swapping allows resources (processors and memories) to be partitioned to form substructures which are added and deleted within the inventory of machine power, in response to environment demand.

[h] The functions of both the Decision Unit and the Control Unit in imposing concurrency control mechanisms (which exist already in the Computation Space net) have eliminated the need for any special concurrency control mechanism on the data.

[i] The incorporation of queues within the body of the instructions allows for the realization of the "Dynamic Concurrency" concept that was claimed as one of the mixed-flow query processing advantages.

[j] The architecture depends only on existing technology in its design, and shows a great simplicity by its replication of a few simple components.

[k] The architecture can be extended in many different ways to incorporate a hierarchy of memories and form multi-ring structures, as will be shown in the next section.

[l] Finally, the data-driven nature of the machine allows the machine units to communicate (asynchronously) by transmitting fixed size information packets. The machine is organized such that these units can tolerate delays in packet transmission without compromising effective utilization of the hardware (notice that DIRECT cannot). This means all units of the machine can work simultaneously: the Page Management Unit pre-delivers the most active data pages into the caches while several operation processing stations are working on their queues. Meanwhile, the Process Scheduling Unit calculates the optimal strategy for managing the current Computation net and at the same time, the query translator splits queries into their elementary transactions and builds their query nets.

6.2. Future work

Future work is this area is likely to focus on two main extensions to the proposed design, namely, a multi-ring architecture extension and a multi-memory expert database machine extension.

The major improvement sought from the multi-ring extension is to improve the performance of the machine in handling very large data files. In this extension several copies of the basic design, each a dedicated ring for performing a specific type of relational database primitive, are intersected in both the Query translator and the switch (see figure 5.2). The machine's basic execution cycle can be adapted using instructions' iteration number, operation code, and query identifier to distribute both instructions and tokens (data and control) from different parts of the computation across these parallel rings. The Page management unit with its buffering subsets is likely to be a common component in order to allow different ready instructions from different machine rings to access their operands.

Beyond the homogenity of traffic within each ring and the feasibility of exploring a VLSI design through a machine organization that consists of identical complex functional units connected together in a regular structure with little off-chip communications, the proposed extension is an excellent solution for improving the performance of very large databases on the proposed machine (see Section 3.5.2.1).

The extension of having a multi-level instruction memory such that the current instruction memory acts as a cache for a larger back-end memory, might provide a chance for having an expert database machine. In essence, the back-end memory acts as a sort of a knowledge base memory, in which the machine stores query nets of different systems' activities (storing the compiled queries) and calls them when needed. This saves the time taken in translation and optimization of query nets.

References

[AnD83] V. Antonellis and B. Demo, Requirements Collection and Analysis of DATAID-1 Project, in Methodology and tools for database design, , 1983.

[Bab79] E. Babb, Implementing a Relational Database by Means of specialized Hardware, *ACM Trans. Database Systems 4*, (1979), 1.

[Bae80] J. Baer, *Computer Systems Architecture*, Computer Science Press, 1980.

[BaH79] J. Banerjee and D.K. Hsiao, Performance Study of a Database Machine in Supporting Relational Databases, *Proc. 4th International Conference on very Large Data Bases*, 1979, 319.

[BeN71] C.G. Bell and A. Newell, *Computer Structures: Readings and Examples*, McGraw-Hill Book, 1971.

[BeY80] Wah W. Benjamin and Bing Yao, DIALOG- A distributed processor organization of database machine, *AFIPS Conference Proceedings 49*, (1980), 243.

[BoD80] H. Boral and D.J. DeWitt, Design consideration for data-flow database machines, *Proc. of the ACM-SIGMOD conference on management of Data (Los Angeles, Calif)*, 1980.

[BoD81] H. Boral and D.J. DeWitt, Processor Allocation Strategies for Multiprocessor Database Machines, *ACM Trans. Database Systems*, 1981, 227.

[BoD83] H. Boral and D.J DeWitt, Database machine..an idea whose time has passed, *3rd International Workshop on Database Machine, Muich*, 1983.

[Bri] Lee Briton, A Description manual for the IDM machine, .

[BCA83] V. Bussolati, S. Ceri, V. De Antonellis and B. Zonta, Views Conceptual Design of DATAID-1 Project, in Methodology and tools for database design, , 1983.

[CHI74] R.H. Canaday, R.D. Harrison, E.L. Ivie, J.L. Ryder and L.A. Wehr, A Back-end Computer for Database Management, *Comm. ACM 17*, (1974), 575.

[ChS75] P.Y. Chang and J.M. Smith, Optimizing the performance of a Relational Algebra Database Interface, *Comm. ACM 18*, (1975), 568.

[ChF80] F. Chin and K.S. Fok, Fast Sorting Algorithms on Uniform Ladders (Multiple Shift Register Loops), *IEEE Trans. on Computers 29*, (1980), 618.

[CLW80] K.M. Chung, F. Luccio and C.K. Wong, On the Complexity of Permuting Records in Magnetic Bubble Memory Systems, *IEEE Trans. on Computers 29*, (1980), 553.

[Cop77] R.G. Copper, The Distributed pipeline, _IEEE Trans. on Computers 26_, (1977), 1123.

[Das81] S. Dasgupta, _S*A Language For Describing Computer Architecture, in Computer Hardware Description Languages and Their Applications_, R. (Ed), North-Holland, 1981.

[Dat83] C.J. Date, _An Introduction to Database Systems, Volume 2_, Addison-Wesley Publishing Company, 1983.

[DeW78] D.J. DeWitt, DIRECT- A Multiprocessor Organization for supporting Relational Database Management systems, _Proc. of the 5th Annual Symposium on Computer Architecture_, 1978, 182.

[DBF80] D.J. DeWitt, H. Boral, D. Friedland and W.K. Wilkinson, Parallel Algorithm For the execution of Relational Database operations, technical Report number 402, University of Wisconsin, 1980.

[DLM77] J.B. Dennis, C.K. Leung and D.P. Misunas, A Highly Parallel Processor Using a Data-flowMachine Language, Computation Structures Group Memo 134-2, Massachustts Institute of Technology, 1977.

[EpH80] Epstein and Hawthorn, Design Decisions for the Intelligent Database Machine, _AFIPS 49_, (1980), 237.

[FeM83] E.A. Feigenbaum and P. McCorduk, _The Fifth Generation Artificial Intelligence and Japan's Computer Challenge to the world._, Computer Sc. Press, 1983.

[FiL77] C.A. Finnila and H.H. Love, The Associative Linear Array Processor, _IEEE Trans. on Computers 26_, (1977), 112.

[FrB79] H.A. Freeman and O.H. Bray, _Data Base Computers_, Lexington Books, D.C. Heath and Company Lexington, Massachusetts Toronto, 1979.

[GPK82] D.D. Garjski, D.A. Padua, D.J. Kuck and R.H. Kuhn, A Second Opinion on Data Flow Machines and languages, _Computers_ , 1982.

[Gel80] E. Gelenbe, Parallel Computation of partial differential equations A modeling approach, _19th IEEE Conf. on Decision & Control_, 1980.

[Gli83] M. Glinz, A Data-flowRetrieval Unit for a Relational Database Machine, _3rd International Workshop on Database Machines, Munich_, 1983.

[GoD80] J.R. Goodman and A.M. Despain, A Study of the interconnection of multiple processors in a database environment, _The 3rd International Conference on Distributed Computing systems_, 1980, 660.

[HaD81] P. Hawthorn and D.J. DeWitt, A performance Evaluation of Database Machine Architectures, _Proc. VLDB-7_, 1981, 199.

[Haz82] A. Hazra, A description Method and a classification Scheme for Data Flow Architecture, _The 3rd International Conference on Distributed Computing systems_, 1982, 645.

[HBB78] D.K. Hsiao, J. Banerjee and R.I. Baum, Concepts and Capabilities of a Database Computer, _Transactions on Computer Systems 3_, (1978), 347.

[Hsi79] D.K. Hsiao, Database Machines are coming database Machines are coming, _Computers 28_, (1979), 7.

[Hsi83] D.K. Hsiao, *Advanced Database Machine Architecture*, Prentice-Hall, Inc., 1983.

[Kin80] W.F. King, Relational Database Systems: Where We Stand Today, *Proc. IFIP Congress* , 1980.

[KIK82] Y. Kiyoka, M. Isoda, K. Kojima, K. Tanaka, A. Minematsu and H. Aiso, Performance Analysis for parallel Processing Schemes of Relational Operations and a Relational Database Machine Architecture with optimal Scheme, *The 3rd, International Conference on Distributed Computing systems*, 1982, 196.

[LeH82] S. Leonard and Haynes, Database Machines Viewed as High Level Language Computers, *Proceedings of the International Workshop on High-level Language Computer Architecture*, 1982.

[LHT82] B. Lubmir, R.L. Hartmann and J. Todhunter, The Active Graph Database Machine, *The 3rd International Conference on Distributed Computing Systems*, 1982, 178.

[MKY81] T.H. Merrett, Y. Kambayashi and H. Yasurra, Scheduling of Page-Fetches in Join operations, *Proceeding VLDB, seventh international conference*, 1981, 488.

[OlB79] E. Oliver and P.B. Berra, The Role of Associative Array Processors in Database Machine Architectures, *Computer 12 3* , (1979), .

[OSS77] E.A. Ozkarahan, S.A. Schuster and K.C. Sevcik, Performance Evaluation of Relational Associative Processor, *ACM Trans. Database Systems*, 1977, 175.

[OzW82] T.M. Ozsu and B. Weide, Modeling of Distributed Database Concurrency Control Mechanisms using An Extended Petri net Formalism, *The 3rd International Conference on Distributed Systems*, 1982, 660.

[RaL77] C.V. Ramamoorthy and H.F. Li, Pipeline Architectures, *Computing Surveys 9* , (1977), 121.

[RGD82] C.V. Ramamoorthy, S.L. Ganesh, S.T. Dong, C.H. Jen and W.T. Tsai, Machines for Data Management in Distributed Systems"" The Design of "Low-End" Machines for Data Management in Distributed Systems, *The 3rd International Conference on Distributed Computing Systems*, 1982, 187.

[RLR79] C. Rolland, R. Leifert and C. Richard, Tools for information system dynamics Management, *Fifth International Conference on Very Large Data Bases*, 1979.

[Rol82] C. Rolland, Petri net model for information management systems, *Proc. VLDB-8*, 1982.

[SNO79] S.A. Schuster, H.B. Nguyen, E. A. Ozkarahan and K.C. Smith, RAP.2- Ann Associative Processor for Database and Its Applications, *IEEE Trans. on Computers 28*, (1979), 446.

[ShZ84] R.K. Shultz and R.J. Zingg, Response Time analysis of Multiprocessor Computers for Database Support, *ACM Trans. Database Systems no. 1*, (1984), 100.

[Sto79] M.R. Stonebraker, *MUFFIN: A Distributed Data Base Machine*, University of California at Berkeley; Electronics Research Laboratory Memo No. UCB/ERL M79/28, 1979.

[SN79] S.Y. Su, H.B. Nguyen, A. Emam and G.J. Lipovski, The Architecture Features and Implementation of the Multi-cell CASSM IEEE Transactions on Computers C-28, , 1979.

[TBH82] P.C. Treleaven, D.R. Brownbridge and R.P. Hapkin, Data-Driven and Demand-Driven Computer Architecture, *ACM Computing Surveys 1 3*, (1982), 93.

[TrL84] P.C. Treleaven and I.G. Lima, Future Computers: Logic, Data-Flow,.......,Control-Flow?, *Computer*, 1984, 47.

[VaG84] P. Valduriez and G. Gardarin, Join and Semijoin Algorithms for a multiprocessor Database Machine, *ACM Trans. Database Systems 9*, (1984), 133.

[Vic81] C.R. Vick, *Dynamic Resources Allocation in Distributed Computing Systems*, UMI Research press, Ann Arbor, Michigan, 1981.

[Yao79] S.B. Yao, Optimization of Query Evaluation Algorithms, *ACM Trans. Database Systems 4*, (1979), 133.

APPENDIX A

An Airline Reservation system Schema

Database Schema:

Passenger(P-number, name, address, Phone)

Booked-on(Site code, flight-no, ok-status, P-number)

Flight(flight-no, flight-capacity, no of available places)

Status-Res(Serial no of Reservation, ok-status)

Res-flight(Serial no of Reservation, flight-no)

Reservation(Serial no of Reservation, P-number)

Departure(date, flight-no, Dest)

We assume that those relations are decomposed as:

R11(P-number, name, address, phone)

R21(Site-code, flight-no, P-number)

R31(Site-code, flight-no, ok-status)

R41(flight-no, flight-capacity)

R42(flight-no, no of available places)

R51(Serial no of Reserv, ok-status)

R61(Serial no of Reserv, flight-no)

R71(Serial no of Reservation, P-number)

R81(date, flight-no, Dest)

The following ia an example of the Ticket Okay activity's elementary transactions:

t_1 Find the person's name who had reserved this Ticket (R71,R11).

t_2 He is the one at the wicket.

t_3 He is not the one at the wicket.

t_4 Find the Ok-status of this reservation (R51).

t_5 This date in nul.

t_6 This date is not equal to nul.

t_7 Find a site-code on that flight-no for the passenger (R61, R31) ; Increase the current capacity (R61,R41). t_8 Assign this site-no to the passenger (R71,R21); Change the ok- status (R51, R31); Destroy entities concerning this reservation (R51, R61, R71).

APPENDIX B

A Proof For Net Maximum-flow Theorem

[Th.1]

If there exists an s-t flow augmenting path L, with respect to a legal flow function f on a net Q, then there exists a legal flow function ,g, on Q such that V(g) > V(f).

Definition of variables

Ps	Page size.
St_i or $St(t_i)$	Selectivity factor of transition t_i
T	Set of net transitions.
P	Set of net places.
∇_i	Amount of augmentation to current flow at place P_i
s,t	Query net source and sink transition(s).
mp_i	Flow potential value at place P_i
n	Path length
PF(P_i)	Popping factor at place P_i

Let $L = P_1 t_1 P_2 t_2 \ldots\ldots t_{n-1} P_n t_n$ ($n \geq 2$) be an s-t flow augmenting path with respect to a legal flow function f on a net Q. It flows from the definitions that $P_1 = s$, $t_n = t$ and t_i is useful from P_i to P_{i+1}, for each i=1,2,....,n-1. More explicitly:

$$f(p_i) < C(P_i) - C(P_i) - f(P_i) > 0 \qquad (1)$$

for each $P_i \epsilon L$

We further assume that

$$Ps * St_i \leq 1 \qquad \text{for all } t_i \epsilon T \qquad (2)$$

Now construct the function g as follows:

[1] Define for each P_i ∊ L a non-zero integer called a popping factor (PF) which represents the number of pages that are required to contribute in achieving a

progress of one page (a progress is defined to be the number of pages at In(t)). These popping factors could be calculated by the equations:

$$PF(In(t)) = 1$$

Where t is the net sink transition

$$PF(P_i) = Int\left(\frac{PF(Pi+1)}{Ps * St(In(Pi+1))}\right) \tag{3}$$

[2] Now define for each $P_i \in$ L, a flow potential value, mp_i, given by the equation:

$$mp_i = Int\left(\frac{C(Pi) - f(Pi)}{PF(Pi)}\right) \tag{4}$$

[3] Identify a place $P_r \in$ L with minimum non-zero flow potential over path L places to be the bottleneck of that path.

$$mp_r \leftarrow \underset{P_i \in L}{Min}(mp_i) \tag{5}$$

[4] For all places P_j having j > r define

$$g(P_{r+i}) = f(P_{r+i}) + \nabla_{r+i} \qquad i = 1,2,\ldots,n-r$$

Where

$$\nabla_{r+i} \leftarrow Int(\nabla_{r+i-1} * St_{r+i-1} * Ps) \tag{6}$$

$$\nabla_r \leftarrow C(P_r) - f(P_r) \tag{7}$$

and

$$\nabla_n \leftarrow mp_r \tag{8}$$

[5] For all places P_j having j < r define

$$g(P_{r-i}) \leftarrow f(P_{r-i}) + \nabla_{r-i} \qquad i = 1,2,\ldots,r-1$$

Where

$$\nabla_{r-i} \leftarrow Int\left(\frac{\nabla_{r-i+1}}{st_{r-i} * Ps}\right) \tag{9}$$

[6] For place P_r

$$g(P_r)=f(P_r)+\nabla_r \tag{10}$$

[7] For other places $P_i \in$ L

$$g(P_i)=f(P_i) \tag{11}$$

It must now be shown that g is a legal flow function on Q and that V(g) > V(f). [Notice that g is defined on all places of Q].

To show that g is a legal flow function on Q, we must simply prove that g satisfies the four conditions of definition 4.8. Since f is a legal flow on Q, it follows that

$$0 \leq f(P_i) \leq C(P_i) \qquad \text{for all } P_i \in P \tag{12}$$

Notice from (2) & (3) that

$$PF(P_1) \geq PF(P_2) \geq \ldots\ldots\ldots \geq PF(In(t))=1 \tag{13}$$

Also from (1),(4) and (13)

$$mp_i > 0 \qquad \text{for all } P_i \in L \tag{14}$$

from (2),(3),(6),(7),(9) and (13)

$$\nabla_1 \geq \nabla_2 \geq \ldots\ldots\ldots \geq \nabla_n > 0 \tag{15}$$

Combining (6),(9) and (15), we have

$$g(P_i) > f(P_i) \qquad \text{for all } P_i \in P \tag{16}$$

Combining (11),(12) and (16), we have

$$0 \leq g(P_i) \qquad \text{for all } P_i \in P \tag{17}$$

Now to prove that g(P_i) <= C(P_i) for all $P_i \in$ P, we have to look at three cases: (a) For the bottleneck P_r (b) For places P_j having j>r (c) For places P_j having j<r

(a) For the bottleneck

Since f is a legal flow then f(P_r) $\leq$ C(P_r), hence all what we have to prove is that $\nabla_r \leq$ C(P_r) - f(P_r)

From (7), we have $\nabla_r = C(P_r) - F(P_r)$ i.e, $\nabla_r \le c(P_r) - f(P_r)$

(b) For j>r

We need to prove that

$$\nabla_{r+i} \le C(P_{r+i}) - f(P_{r+i}) \qquad i=1,2,\ldots\ldots,n-r$$

From (4)

$$mp_{r+i} = Int\left(\frac{C(Pr+i) - f(Pr+i)}{PF(Pr+i)}\right) \tag{18}$$

Combining (13) and (18) we get

$$PF(P_{r+i}) * mp_{r+i} \le C(P_{r+i}) - f(P_{r+i}) \tag{19}$$

From (5)

$$mp_r \le mp_{r+i}$$

i.e;

$$mp_{r+i} \ge \nabla_n \tag{20}$$

From (2) and (6)

$$\nabla_{r+i} \le \nabla_r \tag{21}$$

From (3) and (13),we have

$$\nabla_{r-i} = \nabla_n * PF(P_{r-i}) \quad \& \quad \nabla_{r+i} = \nabla_n * PF(P_{r+i})$$

(22)

Combining (20) and (22) ,we have

$$\nabla_{r+i} \le mp_{r+i} * PF(P_{r+i}) \tag{23}$$

Combining (19) and (23), we have

$$\nabla_{r+i} \le C(P_{r+i}) - f(P_{r+i})$$

(c) For j<r

We need to prove

$$\nabla_{r-i} \leq C(p_{r-i}) - f(P_{r-i}) \qquad i = 1,2,\ldots\ldots,r-1$$

from (4), we have

$$mp_{r-i} = Int\left(\frac{C(Pr-i) - f(Pr-i)}{PF(Pr-i)}\right) \tag{24}$$

From (1), (5), we get

$$PF(P_{r-i}) * \nabla_n \leq mp_{r-i} * PF(p_{r-i}) \tag{25}$$

From (22) and (25), we have

$$\nabla_{r-i} \leq mp_{r-i} * PF(P_{r-i}) \tag{26}$$

Now to get non-recursive equation for ∇_{r-i} , consider the following approximation

$$\nabla_{r-i} = Int\left(\frac{\nabla_r}{(Str-i*Str-i+1*\ldots\ldots*str-1*(Ps)^i)}\right) \tag{27}$$

Similarly, a non-recursive equation for PF(P_{r-i}) could be written as

$$PF(P_{r-i}) = Int\left(\frac{1}{Str-i*Str-i+1*\ldots\ldots*1*(Ps)^{r+i}}\right) \tag{28}$$

From (24) and (28), we get

$$mp_{r-i} = Int(C(P_{r-i}) - f(P_{r-i}))*(St_{r-i}$$

$$*St_{r-i+1}*\ldots\ldots*1*(Ps)^{r+i}) \tag{29}$$

Using (3) and (29), we get

$$mp_{r-i}*PF(P_{r-i}) = Int((C(P_{r-i}) - f(P_{r-i}))$$

$$*(St_{r-i}*st_{r-i+1}*\ldots\ldots*1$$

$$*(Ps)^{r+i}))*\frac{1}{Str-i*Str-i+1*\ldots\ldots*1*(Ps)^{r+}}$$

$$mp_{r-i}*PF(P_{r-i}) = Int((C(P_{r-i}) - f(P_{r-i})) \tag{30}$$

From (26) and (30), we get

$$Int(C(P_{r-i})-f(P_{r-i}))\geq\nabla_{r-i}$$

then

$$C(P_{r-i})-f(P_{r-i})\geq\nabla_{r-i}$$

The three previous cases complete the proof that

$$g(P_i)\leq C(P_i) \qquad \text{for all } P_i\in L \tag{31}$$

Combining (15),(16) and (31), we get

$$0\leq g(P_i)\leq(C(P_i) \qquad \text{for all } P_i$$

This completes the first condition of Definition 4.8. Obviously Condition (2) is still valid, since the new flow g has no effect on the capacities.

We must now prove that g satisfies condition (3) and (4) of definition 4.8. This is accomplished by considering any transition $t_i \in$ T-{t} , since f is a legal flow function on Q,it follows that equation (3) & (4) are valid:

$$St_i{*}Ps{*}f(P_i)-f(P_j)\leq 0 \tag{32}$$

$$\text{for } P_i\in In(t_i)$$

$$\& \quad P_j\in Out(t_i)$$

now if $t_i \in$ L, then no augmentation occurs to any of the input-output places.

Combining (11),(32), we get

$$St_i{*}Ps{*}g(P_i)-g(P_j)\leq 0$$

$$\text{for } P_i\in In(t_i)$$

$$\& \quad P_j\in Out(t_i)$$

If however, $t_i \in$ L, then since L is acyclic and t_i = t, there must be exactly two distinct places incident upon t_i which are contained in L, namely P_{i-1} and P_i . All other places incident upon t_i are therefore $\in$ L.

Now we have to consider two cases (A) t_i precedes P_r (the bottleneck). (B) t_i comes after P_r

(a) Case (A)

Combining (6), (32)

$$St_{r+1} * Ps * g(P_{r+1}) - g(P_{r+2}) =$$

$$St_{r+1} * Ps * (f(P_{r+1}) + \nabla_{r+1}) -$$

$$f(P_{r+2}) + \nabla_{r+1} * St_{r+1} * Ps)$$

$$= St_{r+1} * Ps * f(P_{r+1}) - f(P_{r+2}) \qquad (33)$$

Combining (32), (33), we get

$$St_{r+1} * Ps * g(P_{r+1}) - g(P_{r+2}) \leq 0$$

In general

$$St_i * Ps * g(P_i) - g(P_j) \leq 0 \qquad \text{for } i > r \qquad (34)$$

(b) Case (B)

Combining (9), (32) we get

$$St_{r-1} * Ps * g(P_{r-1}) - g(P_r) =$$

$$St_{r-1} * Ps * (f(P_{r-1}) + \frac{\nabla_r}{Str - 1 * Ps}) - f(P_r) - \nabla_r$$

$$= St_{r-1} * Ps * f(P_{r-1}) - f(P_r) \qquad (35)$$

Combining (32), (35)

$$St_{r-1} * Ps * g(P_{r-1}) - g(P_r) \leq 0$$

$$St_i * Ps * g(P_i) - g(P_j) \leq 0 \qquad \text{for } i < r \qquad (36)$$

Combining (34), (36) shows that the function g satisfies condition [3] and [4].

This completes the proof that the function g is a legal flow function.

Now to show that V(g) > V(f), we first notice that there is exactly one place incident upon t (the sink) which is contained in the path L, namely P_n

$$V(f)=\sum_{P_i\epsilon In(t)} f(P_i)$$

Combining V(f) with (6),(9), we obtain

$$V(g)=\sum_{P_i\epsilon In(t)} g(P_i)$$

$$=\sum_{P_i\epsilon In(t)} f(P_i)+\nabla$$

$$V(g)=V(f)+\nabla$$

Since $\nabla > 0$ then $V(g) > V(f)$ proved.

We now show another proof for parts (b),(c) in page .

(1) Part(c)

For $i < r$

To prove that

$$\nabla_{r-i}\leq C(P_{r-i})-f(P_{r-i}) \qquad i=1,2,\ldots,r\text{-}1$$

From (3), (13), we get

$$\nabla_{r-i}=\nabla_n * PF(P_{r-i}) \tag{37}$$

from (4)

$$mp_{r-i}=Int\left(\frac{C(Pr-i)-f(Pr-i)}{PF(Pr-i)}\right)$$

$$PF(P_{r-i}) * mp_{r-i}\leq C(P_{r-i})-f(P_{r-i}) \tag{38}$$

From (5)

$$mp_{r-i}\geq mp_r \quad \& \quad mp_{r-i}\geq \nabla_n \tag{39}$$

From (39) and (37),we have

$$PF(P_{r-i}) * mp_{r-i}\geq \nabla_{r-i} \tag{40}$$

From (40), (38)

$$C(P_{r-i})-f(P_{r-i})\geq \nabla_{r-i} \qquad \text{Proved}$$

(2) Case (b)

For i > r

From (3) and (13), we have

$$\nabla_r = \nabla_n * PF(P_r)$$

$$\nabla_{r+i} = \nabla_n * PF(P_{r+i}) \qquad (42)$$

Where $\nabla_n = mp_r$ and $PF(P_n) = 1$

From (4)

$$mp_{r+i} = Int\left(\frac{C(Pr+i) - f(Pr+i)}{PF(Pr+i)}\right)$$

According to the definition of Int function, we have

$$PF(P_{r+i}) * mp_{r+i} \leq C(P_{r+i}) - f(P_{r+i}) \qquad (43)$$

From (5)

$$mp_{r+i} \geq mp_r \quad \& \quad mp_{r+i} \geq \nabla_n \qquad (44)$$

From (44) and (42), we have

$$PF(P_{r+i}) * mp_{r+i} \geq \nabla_{r+i} \qquad (45)$$

From (45),(43), we have

$$C(P_{r+i}) - f(P_{r+i}) \geq \nabla_{r+i} \qquad \text{Proved}$$

Note:In case of equation (32) we have incorporated the factor Ps (Page size) to combine both equations (3) & (4) in one proof. It is obviously that the proof (without Ps) is still valid for each case individually.